

Radio Shack

Cat. No. 62-2075

More TRS-80 Assembly Language Programming

By Bill Barden



**An Advanced Look at Challenging, Efficient Assembly-Language
Programs—A Valuable Tool for A Variety of Applications**

U.S.A. \$5.95

**MORE
TRS-80
ASSEMBLY-LANGUAGE
PROGRAMMING**

by

William Barden, Jr.

Radio Shack®

A DIVISION OF TANDY CORPORATION

© 1982 by Radio Shack, a division of Tandy
Corporation, Fort Worth, Texas 76102

FIRST EDITION
FIRST PRINTING

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number:
Library of Congress Catalog Card Number:

Printed in USA

Preface

This book is a follow-up to Radio Shack's book *TRS-80 Assembly-Language Programming*. In that book we described the architecture of the Z-80 microprocessor used in the TRS-80, presented the instruction set and addressing modes, and gave some examples of common assembly-language programming operations.

More TRS-80 Assembly-Language Programming builds on the material found in the previous book by taking an even more practical look at TRS-80 Model I and III assembly-language programming. It answers such questions as "How do I use the Disk Editor/Assembler?," "Where in memory should I put an assembly-language program?," "What about embedded machine-language code?" and "How do I go about writing and implementing a large assembly-language program?"

We have organized the material in this book into four sections. In the first section, "Using Assembly Language," we review some of the general material we covered in the earlier book and then describe the practicalities of assembling programs in the TRS-80. Among the practicalities we discuss are the operation of both the cassette-based Editor/Assembler and Disk Editor/Assembler. Moving along, we get into the general approaches to executing assembly-language programs, both "stand-alone" assembly-language programs and the "embedded in BASIC" machine-language approach.

Section II, "Assembly-Language Techniques," describes four types of processing that can be implemented in assembly language — "number-crunching;" working with character data; table operations, including sorting and searching; and graphics display processing. We present each type of processing first from a general design standpoint — and then we put that theory into practice, with assembly-language code to illustrate the methods of solving the problems of each type of processing.

Because of the unique structure and design of the TRS-80 hardware there are a number of assembly language techniques that are peculiar to this system. Along that line, we take two more chapters in this section to describe cassette input/output, parallel printer operation, and disk file manage operations. Some of you may be especially interested in the material on disk file operations since assembly-language calls may be made directly to TRSDOS disk file manage routines to read and write random files and perform other operations.

The third section, "Larger Assembly-Language Projects," contains complete listings of two large assembly-language programs. The first of these is a Morse Code Program (MORG), which transmits random or defined Morse code messages through the cassette output port at speeds of 5 to 60 words per minute. The second program is an experiment in artificial intelligence — a tic-tac-toe learning program that starts off being not very bright but learns how to play the game until it is virtually unbeatable. Neither program could be implemented in BASIC to execute in "real-time." Also in this section we discuss the general approach in writing large programs in assembly-language along with an actual case study from a TRS-80 software house.

The fourth and final section contains the appendices covering the Z-80 instruction set grouped in functional order and Z-80 operation code listings.

As the author, I hope you find the material in this book a useful supplement to *TRS-80 Assembly-Language Programming*, but I also hope you will be able to use it in its own right. Assembly-language programs are fast, efficient, and challenging, and help to make the TRS-80 even more of a valuable tool for all types of programming applications.

To my wife Janet, for her encouragement

Table Of Contents	
More TRS-80 Assembly-Language Programming	
Section I: Using Assembly Language	
Chapter One	
ASSEMBLY-LANGUAGE BASICS	
A Typical Assembly Program — How the DECBIN Subroutine Works — DECBIN—Detailed Analysis	9
Chapter Two	
ASSEMBLERS AND ASSEMBLING — EDTASM	
A Look at EDTASM — Source Line Syntax — Constants — Pseudo-Operations —	37
Chapter Three	
ASSEMBLERS AND ASSEMBLING — DISK ASSEMBLER	
Comparison between EDTASM and the Disk Assembler — Disk Files for the Disk Assembler — Using the Editor for Source Files — Relocatable Object File Assembly — Macro Capability — Pseudo-Ops for Program Sections and Conditional Assembly — Pseudo-Ops for Listing Format — Using the Disk Assembler — Loading the Object Modules to Produce a Command File	53
Chapter Four	
LOADING, EXECUTING, AND DEBUGGING ASSEMBLY-LANGUAGE PROGRAMS	
EDTASM System Tapes — System Considerations for EDTASM Object Files — Debugging with T-BUG — Disk Assembler Files — Debugging with DEBUG — Interfacing Assembly-Language and BASIC Programs	75

Chapter Five

EMBEDDED MACHINE CODE IN BASIC

Relocatable Code — Embedded Machine Code by
DATA to Memory — Embedded Machine Code by
CHR\$ String — DATA Values and Dummy Strings
— DATA Values and Array Storage — Passing
Arguments and Multiple Subroutines 99

Section II: Assembly-Language Techniques

Chapter Six

NUMBER CRUNCHING

Addition and Subtraction — Multiplies and Divides
— Signed vs. Unsigned Multiplies and Divides —
Overflow Limits — Random Number Generation —
Towards Infinite Precision 113

Chapter Seven

WORKING WITH CHARACTER DATA

Keyboard Operation and Scanning — A Typical
Keyboard Subroutine — Input Subroutines —
Display of Characters — Displaying a Message —
Displaying an Input Character — Scrolling —
Conversion From ASCII Decimal to Binary —
Converting From Binary to Decimal ASCII 135

Chapter Eight

WORKING WITH TABLES

What are Tables? — Fixed Length Entry/Fixed
Length Table — Fixed Length Entry/Variable
Length Table — Variable Length Entry/Variable
Length Table — Jump Tables — Scanning —
Ordered Tables — Searching — Sorting 159

Chapter Nine

GRAPHICS DISPLAY PROCESSING

- Graphics Characteristics — Random vs. Character
- Position Graphics — Character-Oriented Graphics
- Drawing Random Points — Animation — Line
- Drawing 185

Chapter Ten

CASSETTE OUTPUT, MUSIC, and PARALLEL PRINTERS

- Input/Output Programming — Z-80 and TRS-80
- Input/Output — Parallel Printer Operation —
- Cassette I/O 211

Chapter Eleven

DISK I/O IN ASSEMBLY LANGUAGE

- Diskette and Disk Characteristics — Disk Drives —
- The Disk Controller — TRSDOS Disk Organization
- Device Control Blocks — TRSDOS I/O Calls 235

Section III: Larger Assembly-Language Projects

Chapter Twelve

ASSEMBLY-LANGUAGE DESIGN, CODING, AND DEBUGGING

- The Inception Phase — Research — The
- Preliminary Specification — Program Design —
- Coding — Desk Checking — Debugging —
- Comprehensive Checking — Final Clean-Up — No
- Resemblance to Programmers Living or Dead 265

Chapter Thirteen

A MORSE CODE GENERATOR PROGRAM (MORG)

- General Specification — Operation — General
- Design — Implementation — Program Description
- Using This Program 285

Chapter Fourteen

TIC-TAC-TOE LEARNING PROGRAM

General Specifications — Operation — General
Design — Algorithms for playing the Game —
Implementation — Program Description — Using
This Program 333

Appendices

Appendix I. Z-80 INSTRUCTION SET 411

Appendix II. Z-80 OPERATION CODE LISTINGS 415

SECTION I

Using Assembly Language

Chapter One

Assembly-Language Basics

In this chapter we'll review some assembly-language basics. If you want a more comprehensive treatment, refer to *TRS-80 Assembly-Language Programming* (Radio Shack 62-2006). If you've never experimented with assembly language, review some of the material presented in the previous book (especially the "General Concepts" section). An alternative is to read this chapter thoroughly and pay close attention to assembly-language examples we've included later in the book.

By the way, when we use the term "TRS-80" in this book we'll be referring to both the Model I and III. There are some minimal differences between the two systems (which we'll note) but in general, material in this book applies to either system.

The TRS-80 uses the built-in instruction set of the Z-80A microprocessor. The Z-80A is a third-generation microprocessor chip that is truly a "computer on a chip." Using assembly language is mainly a matter of learning both the instruction set of the Z-80A microprocessor and the skills needed in putting together those instructions to form programs.

There are over 700 separate instructions for the Z-80 with a number of different addressing modes. Each of the instructions, however, performs a simple function and is

easy to understand by itself. Furthermore, many of the instructions are similar in nature, and understanding how one specific instruction works might mean that you can easily unravel 23 similar ones!

The instruction set of the Z-80 is provided in Appendix I with the instructions grouped according to logical function. If you want to find an instruction that will load the A register in the Z-80 with a memory operand, for example, you could look under "Loads" and then under "A Load Memory Operand" to find the instructions that perform this function. Many times you'll find a number of different instructions can accomplish the same function.

Appendix II gives the actual format of each instruction. Normally, you will not be concerned with the format, as the Assembler will automatically translate a **mnemonic** for the instruction into the **machine-language** format.

It's possible to translate a string of assembly-language instructions by hand into the equivalent bits shown in the instruction formats of Appendix II. If you did this, the result would be called a **machine-language program**. However, there are a number of excellent automatic **assembler programs** that Radio Shack provides that eliminate such tedious hand work. One of these is the cassette-based Editor/Assembler which takes **symbolic source** lines and translates them into machine-language code. The second is the disk-based Editor/Assembler, a more advanced version of an assembler. We'll look at both assemblers in this book and point out any differences that occur between them.

A Typical Assembly Program

One listing is said to be worth a thousand words . . . Let's take a look at a typical assembly-language program and scrutinize it in some detail to review some of the things we should know before we go on to advanced topics.

The assembly-program **listing** is shown in Figure 1-1. The program itself is designed to convert a character string

representing a decimal value of 0-65535 into an equivalent 16-bit binary value. This program could be used, for example, to convert a keyboard input to binary for processing.

```

08970 ;*****DECIMAL TO BINARY CONVERSION SUBROUTINE*****
08980 ;* CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING *
08990 ;* DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535. *
09000 ;* ENTRY: (HL)=BUFFER CONTAINING ASCII *
09010 ;* (B)=NUMBER OF CHARACTERS *
09020 ;* EXIT: (HL)=BINARY # 0-65535 *
09030 ;* NZ IF INVALID ASCII CHARACTER OTHERWISE Z *
09040 ;* ALL REGISTERS SAVED EXCEPT A,HL *
09050 ;
8590 C5 09060 DECBIN PUSH BC ;SAVE REGISTERS
8591 D5 09070 PUSH DE
8592 DDE5 09080 PUSH IX
8594 DD210000 09090 LD IX,0 ;SET RESULT
8598 DD29 09100 DEC040 ADD IX,IX ;INTERMEDIATE*2
859A DDE5 09110 PUSH IX
859C DD29 09120 ADD IX,IX ;#4
859E DD29 09130 ADD IX,IX ;#8
85A0 D1 09140 POP DE ;#2
85A1 DD19 09150 ADD IX,DE ;#10
85A3 7E 09160 LD A,(HL) ;GET CHARACTER
85A4 D630 09170 SOB 30H ;CONVERT
85A6 FAB685 09180 JP M,DEC070 ;GO IF LT "0"
85A9 FEOA 09190 CP 10 ;TEST FOR GT "9"
85AB F2B685 09200 JP P,DEC070 ;GO IF GT "9"
85AE 5F 09210 LD E,A ;NOW IN E
85AF 1600 09220 LD D,0 ;NOW IN DE
85B1 DD19 09230 ADD IX,DE ;MERGE
85B3 23 09240 INC HL
85B4 10E2 09250 DJNZ DEC040 ;GO IF MORE
85B6 78 09260 LD A,B ;COUNT TO A
85B7 B7 09270 OR A ;SET OR RESET Z FLAG
85B8 DDE5 09280 POSH IX ;RESULT TO HL
85BA E1 09290 POP HL
85BB DDE1 09300 POP IX ;RESTORE REGISTERS
85BD D1 09310 POP DE
85BE C1 09320 POP BC
85BF C9 09330 RET ;RETURN

```

Figure 1-1. Typical Assembly-Language Program

This collection of assembly-language statements makes up a **subroutine**, a structure very similar to a BASIC subroutine. It is located at one place in memory and can be **called** as often as necessary.

There are three main segments of the assembly-language program: the assembly-language **source code**, the edit line numbers and the assembly-language **machine code**.

Figure 1-2 shows the assembly-language source code portion of the listing. The source code consists of symbolic lines similar to BASIC statement lines. In general, each line represents one assembly-language instruction written in **mnemonic** form. The mnemonic of the instruction is

merely a shorthand way to express the instruction. It's much simpler to write DEC DE than to write "take the contents of the DE register, subtract one, and put the results of the operation back into DE".

	LABEL FIELD	OP-CODE FIELD	OPERAND FIELD	COMMENTS FIELD
	08970	*****	DECIMAL TO BINARY CONVERSION SUBROUTINE*****	
	08980	;	CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING	
	08990	;	DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.	
	09000	;	ENTRY: (HL)=BUFFER CONTAINING ASCII	
	09010	;	(B)=NUMBER OF CHARACTERS	
	09020	;	EXIT: (HL)=BINARY # D-65535	
	09030	;	NZ IF INVALID ASCII CHARACTER OTHERWISE Z	
	09040	;	ALL REGISTERS SAVED EXCEPT A,HL	
	09050			
859D	C5	09060	DEC BIN PUSH BC	;SAVE REGISTERS
8591	D5	09070	PUSH DE	
8592	DDE5	09080	PUSH IX	
8594	DD210000	09090	LD IX,0	;SET RESULT
8598	DD29	09100	ADD IX,IX	;INTERMEDIATE*2
859A	DDE5	09110	PUSH IX	
859C	DD29	09120	ADD IX,IX	;#4
859E	DD29	09130	ADD IX,IX	;#8
85A0	D1	09140	POP DE	;#2
85A1	DD19	09150	ADD IX,DE	;#1D
85A3	7E	09160	LD A,(HL)	;GET CHARACTER
85A4	D630	09170	SUB 3DH	;CONVERT
85A6	FAB685	09180	JP *M,DECD7D	;GO IF LT "0"
85A9	FE0A	09190	CP 10	;TEST FOR GT "9"
85AB	F2B685	09200	JP P,DECD70	;GO IF GT "9"
85AE	5F	09210	LD E,A	;NOW IN E
85AF	1600	09220	LD D,D	;NOW IN DE
85B1	DD19	09230	ADD IX,DE	;MERGE
85B3	23	09240	INC HL	
85B4	10E2	09250	DJNZ DECD4D	;GO IF MORE
85B6	78	09260	LD A,B	;COUNT TO A
85B7	B7	09270	OR A	;SET OR RESET Z FLAG
85B8	DDE5	09280	PUSH IX	;RESULT TO HL
85BA	E1	09290	POP HL	
85BB	DDE1	09300	POP IX	;RESTORE REGISTERS
85BD	D1	09310	POP DE	
85BE	C1	09320	POP BC	
85BF	C9	09330	RET	;RETURN

Figure 1-2. Source Code

There are four fields in each assembly-language source line.

The second field is the mnemonic representing the instruction to be used. Each of the 700 or so instructions has a predefined mnemonic that the assembler recognizes as a valid instruction. As the mnemonic defines an **operation code** for the instruction, this field is often referred to as the **op-code** field.

The third field in the source line is the **operand** field. The number of operands for instructions varies from none to three. The RET instruction shown in the figure, for

example, requires no operands, while one of the LD instructions requires two — one specifying the register pair that points to a memory address $\{ (HL) \}$, and a second specifying the register to be loaded with the contents of that memory address (A) . The complete op-code and operand form of the instruction is `LD A, (HL)`.

The fourth field of the instruction is an optional **comments** field. This field is used solely for comments associated with the instruction, similar to the “REMARKS” statement of BASIC. A source line is a **comment line** when the line starts with a semicolon (;). *The comments field must always start with a semicolon.*

The remaining field of the source line is the optional **label field**. We use this field to define a label for the instruction, which can then be referenced by another instruction in the program. The line `DEC040 ADD IX, IX`, for example, has the label `DEC040` for the `ADD` instruction. A later instruction, `DJNZ DEC040`, causes a jump to `DEC040` if there is a zero result in the B register. The use of symbolic names for instruction locations makes it unnecessary to keep track of the absolute locations for each instruction in memory. Keeping track of absolute locations is a burdensome chore, although it can be done in **machine-language programming**, which does not use an assembler. The label may or may not be suffixed by a colon, depending upon the assembler.

The second part of the assembly listing, shown in Figure 1-3, is the **editing of line numbers**. The source code lines are entered from the keyboard to an **editor program**, which is usually part of an Editor/Assembler package. The Editor uses line numbers for each of the source lines with the line numbers in ascending sequence. The typical line number increment is 10, so that the line numbers are 100, 110, 120, etc. Using the Editor, lines may be modified, inserted, or deleted. Characters within the lines may also be processed.

```

08970 ;*****DECIMAL TO BINARY CONVERSION SUBROUTINE*****
08980 ; CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING
08990 ; DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.
09000 ; ENTRY: (HL)=BUFFER CONTAINING ASCII
09010 ; (B)=NUMBER OF CHARACTERS
09020 ; EXIT: (HL)=BINARY # 0-65535
09030 ; NZ IF INVALID ASCII CHARACTER OTHERWISE Z
09040 ; ALL REGISTERS SAVED EXCEPT A,HL
09050 ;
8590 C5 09060 DECBIN PUSH BC ;SAVE REGISTERS
8591 D5 09070 PUSH DE
8592 DDE5 09080 PUSH IX
8594 DD210000 09090 LD IX,0 ;SET RESULT
8598 DD29 09100 DECO40 ADD IX,IX ;INTERMEDIATE*2
859A DDE5 09110 PUSH IX
859C DD29 09120 ADD IX,IX ;*4
859E DD29 09130 ADD IX,IX ;*8
85A0 D1 09140 POP DE ;*2
85A1 DD19 09150 ADD IX,DE ;*10
85A3 7E 09160 LD A,(HL) ;GET CHARACTER
85A4 D630 09170 SUB 30H ;CONVERT
85A6 FAB685 09180 JP M,DECO70 ;GO IF LT "0"
85A9 FE0A 09190 CP 10 ;TEST FOR GT "9"
85AB F2B685 09200 JP P,DECO70 ;GO IF GT "9"
85AE 5F 09210 LD E,A ;NOW IN E
85AF 1600 09220 LD D,0 ;NOW IN DE
85B1 DD19 09230 ADD IX,DE ;MERGE
85B3 23 09240 INC HL
85B4 10E2 09250 DJNZ DECO40 ;GO IF MORE
85B6 78 09260 DECO70 LD A,B ;COUNT TO A
85B7 B7 09270 OR A ;SET OR RESET Z FLAG
85B8 DDE5 09280 PUSH IX ;RESULT TO HL
85BA E1 09290 POP HL
85BB DDE1 09300 POP IX ;RESTORE REGISTERS
85BD D1 09310 POP DE
85BE C1 09320 POP BC
85BF C9 09330 RET ;RETURN

```

Figure 1-3. Editing Line Numbers

It's important to note that the edit line numbers are used for editing purposes only and are not used in the program (as they are in BASIC) to refer to other source lines. The output of the Editor is an assembly-language **source file**, a collection of source lines that represent a source program. This file is stored on cassette tape or disk, depending upon the Editor/Assembler.

The third part of the assembly-language listing is shown in Figure 1-4. This is the **machine code** portion of the assembly. When the Assembler portion of the Editor/Assembler processes the source file, the Assembler **translates** the source lines into the corresponding machine code form of the instruction. For example, the Assembler translates the OR A . . . instruction into a hexadecimal "B7". The hex B7 corresponds to a binary 10110111, which the Z-80 microprocessor will decode as an OR instruction that ORs the contents of the A register with the A register. The Assembler automatically translates the symbolic source lines into a form that the Z-80 processor can recognize, binary ones and zeroes.

MEMORY LOCATION
FOR INSTRUCTION

↓

DATA REPRINTING
INSTRUCTION

↓

	08970	*****DECIMAL TO BINARY CONVERSION SUBROUTINE*****	
	08980	CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING	
	08990	DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535.	
	09000	ENTRY: (HL)=BUFFER CONTAINING ASCII	
	09010	(B)=NUMBER OF CHARACTERS	
	09020	EXIT: (HL)=BINARY # 0-65535	
	09030	NZ IF INVALID ASCII CHARACTER OTHERWISE Z	
	09040	ALL REGISTERS SAVED EXCEPT A,HL	
	09050		
8590	C5	09060 DECBIN	PUSH BC ;SAVE REGISTERS
8591	D5	09070	PUSH DE
8592	DDE5	09080	PUSH IX
8594	DD210000	09090	LD IX,0 ;SET RESULT
8598	DD29	09100 DECO40	ADD IX,IX ;INTERMEDIATE*2
859A	00E5	09110	PUSH IX
859C	DD29	09120	ADD IX,IX ;#4
859E	DD29	09130	ADD IX,IX ;#8
85A0	D1	09140	POP DE ;#2
85A1	DD19	09150	ADD IX,DE ;#10
85A3	7E	09160	LD A,(HL) ;GET CHARACTER
85A4	D630	09170	SUB 30H ;CONVERT
85A6	FAB685	09180	JP M,DECO70 ;GO IF LT "0"
85A9	FE0A	09190	CP 10 ;TEST FOR GT "9"
85AB	F2B685	09200	JP P,DECO70 ;GO IF GT "9"
85AE	5F	09210	LD E,A ;NOW IN E
85AF	1600	09220	LD D,0 ;NOW IN DE
85B1	DD19	09230	ADD IX,DE ;MERGE
85B3	23	09240	INC HL
85B4	10E2	09250	DJNZ DECO40 ;DO IF MORE
85B6	78	09260 DECO70	LD A,B ;COUNT TO A
85B7	B7	09270	OR A ;SET OR RESET Z FLAG
85B8	DDE5	09280	PUSH IX ;RESULT TO HL
85BA	E1	09290	POP HL
85BB	DDE1	09300	POP IX ;RESTORE REGISTERS
85BD	D1	09310	POP DE
85BE	C1	09320	POP BC
85BF	C9	09330	RET ;RETURN

Figure 1-4. Machine Code

The second column of Figure 1-4 is the hexadecimal representation of all the assembly source lines. As there are two hexadecimal digits per 8 bits (one byte), each pair of hex digits represents one byte of the instruction. Z-80 instructions may be one, two, three, or four bytes long, as you can see from the figure. Normally, the assembly-language programmer should know the rudiments of binary and hexadecimal representation. If you're not familiar with binary or hexadecimal numbers, spend a few hours reviewing a basic text on the subject.

Hints and Kinks 1-1

Binary/Hexadecimal Representation

Hexadecimal is just a shorthand way to represent binary numbers. To convert from binary to hex.

group the binary number into 4-bit groups and then convert to a hex digit 0-9, A, B, C, D, E, or F.

Binary	0011	1011	1001	1111
			↓	
Hex	3	B	9	F

To convert back, perform the operation in reverse.

To convert from decimal to binary, divide by two and arrange the remainders in reverse order.

0	RI	} 1101101
2 1	RI	
2 3	R0	
2 6	RI	
2 13	RI	
2 27	R0	
2 54	RI	
2 109		←

Use the same scheme to convert from decimal to hexadecimal.

To convert from binary to decimal, use "double-dabble."

1101101	$1 \times 2 = 2 + 1 = 3$
110110	$3 \times 2 = 6 + 0 = 6$
11011	$6 \times 2 = 12 + 1 = 13$
1101	$13 \times 2 = 26 + 1 = 27$
110	$27 \times 2 = 54 + 0 = 54$
11	$54 \times 2 = 108 + 1 = 109$

Use the same scheme to convert from hex to decimal.

. . . . I know - Right now you're saying, "If God had wanted man to count in hexadecimal, he would have given him 16 fingers"

You'll be surprised, however, at how easy it is to work in binary or hex after some practice. Soon you'll be balancing your checkbook in it! Hmmmm . . . Maybe that would help mine

The first column of the machine code portion, which is in hexadecimal notation, represents the **location** of the machine-code data in memory. The instruction PUSH BC, for example, has been assembled at location 8590H. If you were to take the machine code from column two and enter it into RAM (*Random Access Memory*) starting at location 8590H (by using T-BUG or DEBUG), the 48 bytes from 8590H to 85BFH would represent the machine-language program for the DECBIN subroutine. The location column often represents the **absolute location** of the instructions, although with the Disk Editor/Assembler it may represent the **relative locations** from the start (we'll see how in a later chapter).

How the DECBIN Subroutine Works

Now let's take a look at the operation of DECBIN so we can review some basic concepts about Z-80 architecture, instructions, and addressing. Before we wrote DECBIN, we knew the functions it had to perform — we wanted to write some assembly-language code that would take a given ASCII string representing a decimal number and convert it to binary form.

Subroutines

It's convenient to write this code as a **subroutine**. A subroutine is nothing more than a collection of instructions that performs a particular function. Subroutines are handy — rather than write the instructions each time we need the function, we define the subroutine once so it occupies one particular area in memory, and then we call it up whenever we need that function in the program. This little convenience saves memory space since the code only occupies one point in memory. It also saves us the development time of writing the source lines over and over. Assembly-language subroutines are functionally identical to BASIC subroutines.

Subroutines also are important for another reason. They break the program up into a number of small modules that perform well-defined functions. This pattern of modules

makes the entire programming task much easier than writing a large amount of **in-line** code.

We can call the DECBIN subroutine by a CALL instruction that is very similar to a BASIC GOSUB statement. The CALL instruction **jumps** to the subroutine but saves the return address of the next instruction after the CALL. When the RET, or RETURN instruction in DECBIN is finally executed at completion, the **return address** of the instruction after the CALL is retrieved or **popped** from the **memory stack**, and a return is made.

The appearance of the CALL might be:

```
NAME  CALL  DECBIN ;THIS IS THE CALL TO DECBIN
      LD    A, 1    ;RETURN HERE
      ,
      ,
      ,
```

Stack Operations

The memory stack is an area of RAM set aside for certain functions. Whenever a CALL instruction is executed, the current address of the Z-80 **program counter** register is saved in the stack area; the address is "pushed" onto the stack as shown in Figure 1-5. Whenever a RET instruction is executed, the return address is retrieved, or "popped" from the stack and put into the program counter to cause a return to the instruction following the CALL. The stack is also used as temporary storage, as we shall see. The stack area set aside for this function and others is typically 100 bytes.

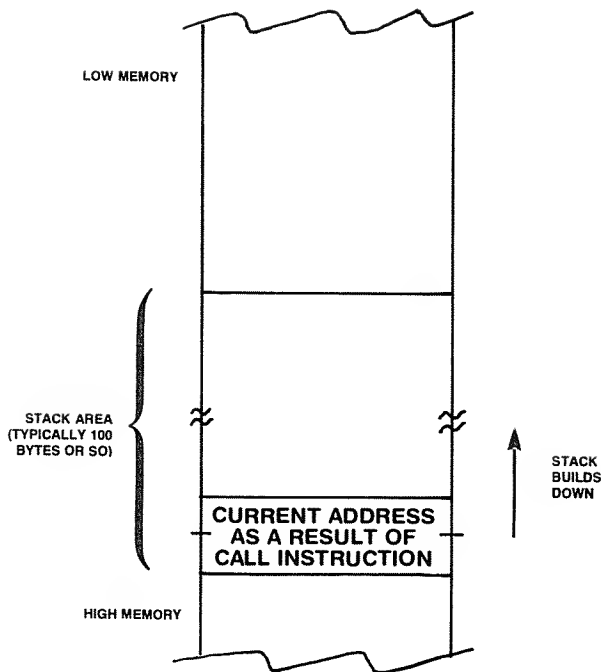
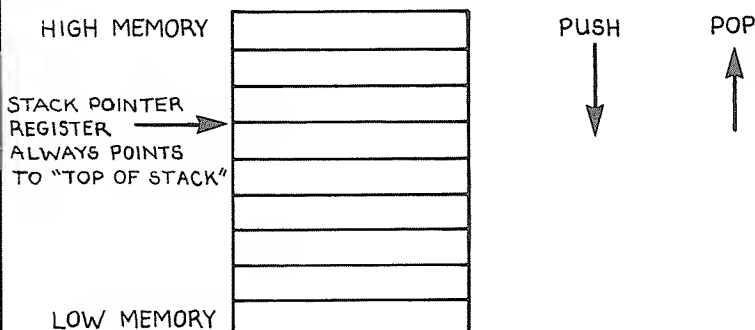


Figure 1-5. Stack Action for a CALL/RET

Hints and Kinks 1-2

Stack use

It's easy to remember how the stack works if you think of a dinner plate stacker in a restaurant. (No, not Ed, the part time counter man - the mechanical one!) As a plate (8 bits of data) is placed on the stack, the stack is pushed down. As a plate (data) is taken off the stack, the stack pops up. Every time a CALL is made, two bytes of data, making up two address bytes, are pushed onto the stack. Every RETURN pops the two bytes. PUSH and POP instructions operate similarly.



To set the stack pointer, perform the "LD SP,XXXX" instruction, where XXXX is some (usually) high memory location, as the stack "builds down".

Just as every he must have a she, every PUSH must have a POP and every CALL must have a RET! Otherwise stack will "gobble up" memory below it as it digests byte after byte, pushing data further and further down!

Before we can call the DECBIN subroutine, we must set up the subroutine **arguments** in preparation for the subroutine action. The arguments are what might be popularly known as the “gozintas” and the “gozoutas.” In this subroutine, we’re **entering** with a pointer to a string of ASCII characters representing decimal digits 0-9 and the number of characters in the string. That’s the “goes into.” We’re **exiting** (the “goes out of”) with a binary number from 0 to 65535 representing the converted result. Also, if we detect an invalid ASCII character in the string, such as the one in “123\$56” — we want to know about it. In the case of “123\$56”, the **Z flag** in the Z-80 is reset (an “NZ” condition).

When we started writing the subroutine, we had to do some thinking about how to pass the arguments. One of the arguments is a pointer to a string of ASCII characters. The characters are identical to the ones we would get by typing in characters from the TRS-80 keyboard. Each character takes up one byte, as shown in Figure 1-6. We would like to convert the string of characters into the equivalent binary number. Since we must put some limit on the number to be converted, we chose 65535, a number that can be held in 16 bits, which is the size of a Z-80 register pair. It seems convenient, then, to return the result in a Z-80 register pair of 16 bits.

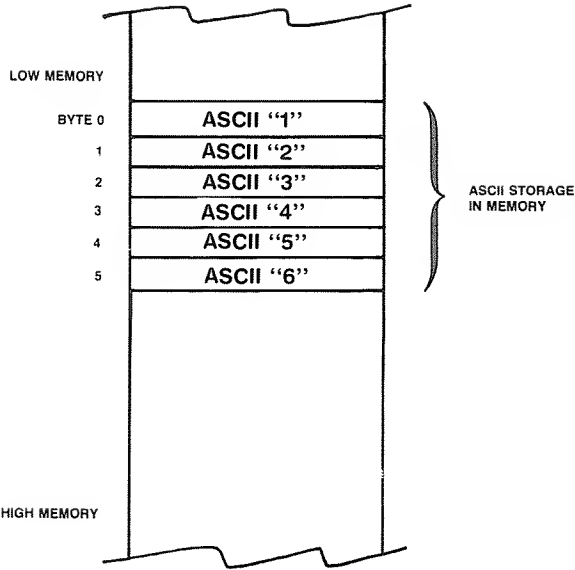


Figure 1-6. ASCII Character Storage

ASCII

ASCII is simply a standardized 7-bit code used for uniform representation of character data. See the tables in many of the TRS-80 manuals. All BASIC strings are stored in ASCII. Data to be displayed on the screen or line printer must first be converted to ASCII by software.

Fortunately the ASCII codes for 0-9 and A-Z and a-z run in sequence, making it much easier to convert. This is not always true of character codes. Some peripherals have codes based on type-ball position, the phase of the moon, or both!

Passing Arguments

How would we pass the ASCII string to the subroutine? With a limit of 65535, we would have a maximum of 5 bytes. Although we could pass these five bytes in five Z-80 registers (or $2\frac{1}{2}$ register pairs), we'll choose instead to pass a **pointer** to the memory locations that contain the string. This pointer value can be held in 16 bits, as any TRS-80 memory location can be expressed as a value of 0 through 65535.

The number of bytes in the string may vary from 1 to 5; this is also an argument that must be passed to the DECBIN subroutine. As this is a small value, it can be passed in a single Z-80 register of 8 bits.

Z-80 Registers

Any of the general purpose (A, B, C, D, E, H, or L) registers or register pairs (BC, DE, HL) could have been chosen to hold the arguments. We have chosen HL to hold the string pointer and B to hold the count — since HL is convenient because many of the instructions use the HL register pair as a **register indirect pointer** to memory locations. B is

conveniently used, as a special DJNZ instruction uses B as a counter. A typical CALL to the DECBIN might now look like this:

```
NAME1 LD    HL,BUFFER    ;ADDRESS OF
                                STRING BUFFER
      LD    B,5          ;# OF CHARACTERS
      CALL DECBIN        ;CONVERT ASCII
                                TO BINARY
      LD    A,1          ;RETURN HERE
      .
      .
      .
```

Z-80 Registers

These are the Z-80 registers accessible to TRS-80 assembly-language programmers. A is used for many arithmetic operations. The HL register pair is used as a '16-bit accumulator' for 16-bit arithmetic. Selection of either AF or AF' is done by EX AF,AF'. Selection of either B-L or B'-L' is done by EXX.

Seem like it is easy to get confused about which set of registers, prime or non-prime you are using? You bet. Much TRS-80 software uses only one set. Typically the prime set would be used for interrupt processing. There is no reason that you should not use both sets, however, as long as you keep track of which set is 'current.'

REGISTER PAIRS	16 BITS		16 BITS		
	8 BITS	8 BITS	8 BITS	8 BITS	
AF	A	F	A'	F'	GENERAL-PURPOSE REGISTER (ONLY ONE SET ACTIVE, EITHER A-L OR A'-L')
BC	B	C	B'	C'	
DE	D	E	D'	E'	
HL	H	L	H'	L'	
	IX		INDEX REGISTER (USED IN INDEXED ADDRESSING)		
	IY		INDEX REGISTER (" " " ")		
	PC		PROGRAM COUNTER (KEEPS TRACK OF CURRENT INSTRUCTION)		
	SP		STACK POINTER (POINTS TO TOP OF STACK AREA)		
	I	R	INTERRUPT REGISTER } NOT NORMALLY REFRESH REGISTER } USED IN TRS-80 PROGRAMMING		

In the above sequence, the HL register pair is loaded with the address of the string **buffer**, the locations at which the ASCII characters are stored. Note that the LD HL,XX instruction (see Appendix II) is an **immediate-load** type

instruction that loads a 16-bit value into the HL register pair. In this case, the assembler will load a 16-bit address corresponding to the location of label **BUFFER** into HL. We have loaded the B register with 5 as a typical string length. The **CALL** is then made to the **DECBIN** subroutine, with a return after the conversion to the instruction following the **CALL**.

Algorithm for DECBIN

Now let's get into the operation of **DECBIN** itself. The sequence of operations for **DECBIN** goes something like this:

1. Clear a result variable.
2. Get an ASCII digit from memory, starting from the leftmost digit of the string.
3. Convert the ASCII digit to binary. (Since an ASCII "0" through "9" is represented by hexadecimal 30 through 39, this involves subtracting 30H from the ASCII digit.)
4. Add the result of the subtract, binary 0-9, to the total.
5. If this is the last ASCII digit, you're done. If it's not the last digit, multiply by 10 and go back to step 2.

We've used a slightly modified version of this **algorithm** in **DECBIN**. Rather than checking for the last ASCII digit and **then** multiplying by 10, we'll multiply by 10 as an initial action. This process is illustrated in Figure 1-7, for an ASCII string of 5 digits.

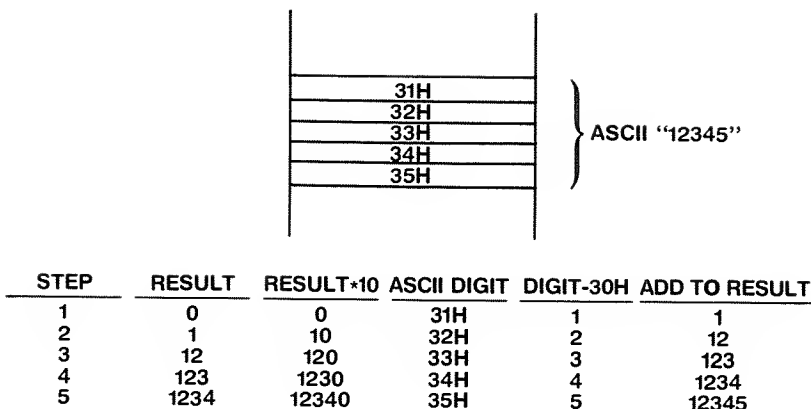


Figure 1-7. Algorithm for DECIBIN

DECIBIN — Detailed Analysis

Now let's go through the DECIBIN subroutine to see how this is implemented in code. The first nine lines of the subroutine are comment lines describing the action of the subroutine, the entry conditions, and the exit conditions. The remainder of the subroutine implements the algorithm.

The name of the subroutine DECIBIN, is expressed by the label opposite the first instruction mnemonic. During assembly time, this label will be equated to the location of the subroutine, and any CALL DECIBIN will assemble as a CALL to the proper absolute location.

Symbol Table

As the Assembler does its work, it builds a symbol table in memory. Every symbol encountered is put into the table as a six-character name and associated address value. Any references to the symbol cause the Assembler to search the symbol table for the value of the symbol.

SYMBOL	VALUE	
DEC 020	8000	← USUALLY CORRESPONDS TO LOCATION OF LABEL IN MEMORY
DEC 040	8XXX	
DEC 070	8XXX	
EDD 000	FFFF	← AN "UNRESOLVED SYMBOL" NOT YET ENCOUNTERED IN SOURCE LINE OR COULD ALSO BE "UNDEFINED"
EDD 070	9XXX	
ENN 000	9AAF	
FOO 000	7XXX	← A... OH, NEVER MIND...
SEX 000	9ABC	

The first three instructions PUSH the BC and DE register pairs and the IX register onto the stack, which allows the stack to be used as a temporary storage area. The contents of BC, DE, and IX, six bytes in all, will be saved in the stack area as shown in figure 1-8. They will be retrieved with subsequent POP instructions. "Why," you ask, "are the contents of BC, DE, and IX saved?"

The answer is that it's convenient for the programmer to CALL subroutines with data in the Z-80 general purpose or other registers without having to worry about those

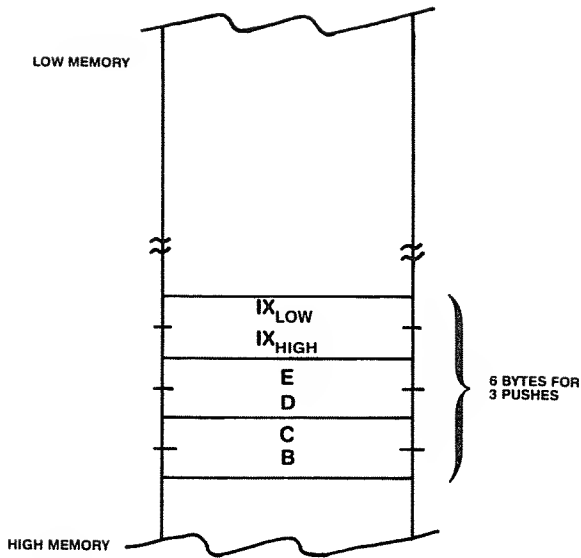


Figure 1-8. Stack Action for PUSH/POP

registers being used and the data being destroyed. As B, DE, and IX are used in the processing of DECBIN, their initial contents are saved in the stack at the beginning of the subroutine. Now the registers can be used for calculations and intermediate results during the processing of the subroutine, and their contents can be restored just prior to the RETurn. Of course any registers that are used to **pass back** the results of the subroutine would have new contents anyway and should not be saved in the stack.

The LD IX, 0 instruction clears the IX index register to 0. IX will hold the intermediate results of the conversion.

The assembly-language code from label DEC040 through DJNZ DEC040, . represents the main body of the subroutine. It is a loop (and the remarks are indented to indicate the loop condition). The number of times through

the loop is determined by the number of ASCII digits to be processed. Five digits would involve five iterations, four would be four iterations, and so on.

The first order of business is to take the current contents of the IX register and to multiply it by 10. Although we could CALL another subroutine that performs a multiply, we choose to do it an alternative way by a "shift and add" technique. First we perform the `ADD IX,IX`, which adds the contents of the IX register to itself. Any number added to itself doubles the number, and that's the case here. The `number*2` is then is PUSHed into the stack by the `PUSH IX` instruction. After the `PUSH`, the `number*2` is in both IX and the stack. Two more `ADDS` change the contents of IX to `number*4` and `number*8`, respectively. The next instruction, `POP DE`, pops the `number*2` value from the stack and puts it into register pair DE. Now an `ADD` of IX and DE is done with the result going to IX. We are adding (`number*8 + number*2`), which is the same as `number*10`. In effect, we've multiplied the contents of IX by 10.

The `LD A,(HL)` instruction is a Load instruction that loads the A register with the contents of a memory location. The HL register pair is used as a **register indirect** pointer to the memory location to be loaded. If, for example, HL contained 9000H, the contents of memory location 9000H would be loaded into the A register. The parentheses around the (HL) indicate that a **memory location**, rather than an **immediate** value, is involved in the instruction.

As we entered the subroutine with the HL register pointing to the first (leftmost) byte of the string, the first execution of `LD A,(HL)` would result in the first ASCII character of the string being loaded into the A register.

The ASCII character in A is now converted from hex 30 through 39 (ASCII "0" through "9") to a binary value of 0 through 9 by subtracting 30H. An ASCII "5", for example, would produce a binary value of $35\text{H}-30\text{H}=5$.

There is one problem here, however. If the ASCII character is not 30H through 39H, the result will be incorrect. We chose to check the validity of the ASCII string in the subroutine, so we'd best do that right now. If the result of (ASCII character-30H) is less than 0, then the ASCII character was less than 30H. If the result of (ASCII character-30H) is greater than 9, the ASCII character is greater than 39H.

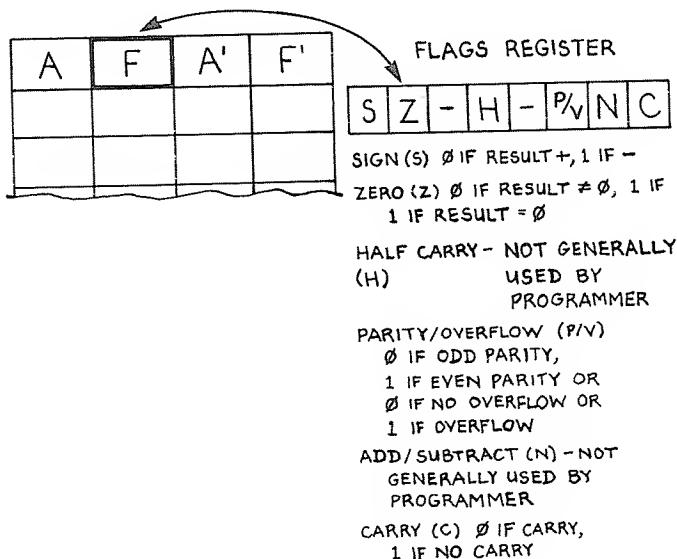
When we performed the `SUB 30H`, the set of **flags** in the Z-80 was set according to the result of the subtract. If the result was 0, the Z flag was set; otherwise, it was reset (NZ). If the result was positive the Sign flag was reset (P); otherwise, it was set (M or minus). Similarly, the other flags (half-carry, carry, parity/overflow, add/subtract) were affected. The flags are not always affected by an instruction — LoadS never affect flags, for instance. However, many instructions, especially adds and subtracts, do affect the flags according to the results of the instruction.

Hints and Kinks 1-6

Flags

Flags are a collection of individual bits grouped as the "flags register." Flags are used to test results of instructions by conditional jumps. Some instructions (LDs, PUSH, POP, RET, others) never affect the flags. Other instructions (ADDs, SUBs, CP) always affect the flags while still other instructions only affect certain flags. See Appendix II.

Don't hesitate to use instructions that don't affect the flags before testing the flags with conditional jumps. Just be certain which instructions affect which flags - there are some surprises here!



We can test the result of the subtract for a minus condition by the JP M,DEC070 instruction. If the result of (ASCII character-30H) was negative, the Sign flag will be set (M), and the instruction will jump on the M condition to label

DEC070. If the result was positive, the jump will be ignored, and the next instruction in sequence will be executed. If the result was negative, the jump to DEC070 will cause the subroutine to terminate prematurely based on the invalid ASCII character.

After the **conditional jump**, we make another test. The CP 10 instruction compares the result to 10. *A compare is essentially a subtract of an 8-bit value from the contents of the A register.* Like a subtract, the flags are affected and can be used for conditional branching. Unlike a subtract, the contents of A remain unchanged; the result is discarded. In this case the value to be subtracted is an **immediate value** of 10. If the Sign flag is set after the CP, the conditional jump JP P,DEC070 will cause a jump to location DEC070 to terminate the subroutine because of an invalid ASCII character. Otherwise the next instruction in sequence will be executed.

Assuming that the ASCII character was valid, we now have a binary value of 0-9 in the A register. The LD E,A copies that value from A into the E register. The next instruction, LD D,0, loads the D register with a 0 value. In fact, the DE register pair is now loaded with the value 0-9. If the D register were not cleared by loading 0, the DE register pair would contain garbage in the D register, and DE could not be used in further operations involving the binary value of 0-9.

Now the binary value in DE is added to the intermediate result in IX. If this is the first time through the loop, IX now contains the binary value of 0-9. If this is the second time through the loop, IX contains $A1*10 + A2$, where $A1$ is the first converted ASCII value and $A2$ is the next. The third time produces $A1*100 + A2*10 + A3$, and so on.

The INC HL instruction adds one (INCRements) to the contents of the HL register and puts the results back into HL. (No flags are affected, by the way.) Initially, HL pointed to the leftmost character in the buffer. Each time through the loop, HL is adjusted to point to the next ASCII character so it may be processed in turn.

The DJNZ DEC040 is a unique instruction. It operates as follows: The contents of the B register is decremented by one. If the result of the decrement is not zero (NZ), the instruction jumps back to the specified jump address, in this case DEC040. If the result of the decrement is zero, the next instruction in sequence is executed. The mnemonic stands for "Decrement and Jump if Not Zero."

We entered the DECBIN subroutine with a count in the B register of 1 to 5, representing the number of ASCII digits to be converted. Each time through the loop, the count in B is adjusted downward by one until zero is reached. As long as the contents of B **after the decrement** is not zero, the loop is reentered at DEC040.

— Hints and Kinks 1-7 —

Loop Trace

Here's a "trace" of individual registers through part of the DECBIN routine. This type of trace can be done with paper and pencil to "play computer" and verify during "desk checking" that the program works as desired.

<u>A</u>	<u>IX</u>	<u>DE</u>	<u>HL</u>	<u>B</u>
31	0		BUF	3
1	1	1	BUF+1	2
	2			
	4			
	8	2		
	10			
32				
2	12	2	BUF+2	1
	24			

Right now you're probably asking. "Must assembly-language programming be this tedious?" Well, yes... Look at the rewards. though—fast speed, compact code, the challenge...uh...maybe I'll go back to BASIC...

After the loop has been completed, IX contains the result of the conversion of the ASCII string. The LD A,B instruction loads the A register with the contents of B. If the count in B is 0, the loop has been completed for all ASCII digits. If B is other than 0, a premature jump was made to DEC070 because of an invalid ASCII digit. After B has been loaded to A, we execute an OR A instruction. This instruction is a commonly used instruction to test the contents of A. ORing any number with itself does not change the number. The OR A takes the contents of the specified register (A) and ORs it with the A register. When the specified register is A, this ORs A with itself. The important thing here, though, is that the flags are set on the result of the OR. The Z flag is set if the result in A is zero or reset if the result is non-0. Since A contains the previous count in B, the Z flag is set if B was zero (successful termination of loop) or reset if B was non-zero (invalid ASCII character). As the following PUSH, POP, and RET instructions **do not affect the flags**, the Z flag will remain set or reset on the return to the **calling program**.

As the result is in IX, and we specified the result in HL on exit, a transfer must be made. A PUSH IX followed by a POP HL pushes the contents of IX onto the stack and then immediately pops it back to the HL register pair. This is a common way to transfer the contents of one register pair to another as there is no LD from one register pair to another. The three POPs at the end of the subroutine restore the original contents of IX, DE, and BC. Note that the order of the contents is opposite from the way they were initially pushed onto the stack as the stack is a "last in, first out" operation.

The RET instruction pops the return address from the stack, puts it into the program counter, and causes a return back to the calling program at the next instruction after the CALL DECBIN. The HL register pair now contains the result of the conversion (0-65535), and the Z flag is set if the conversion was correct or reset if an invalid ASCII character was encountered.

The DECBIN subroutine is probably a typical TRS-80 subroutine in terms of complexity and size. If you have trouble with some of the concepts involved with DECBIN, review the appropriate material in *TRS-80 Assembly-Language Programming*, and continue to follow the examples closely in future chapters. We'll attempt to explain any subtleties as they come up.

Chapter Two

Assemblers and Assembling- EDTASM

Radio Shack has two assemblers available for the TRS-80, the cassette-based Editor/Assembler and the disk-based Disk Editor/Assembler. To make things easier, we'll refer to the former as "EDTASM" and the latter as "the Disk Assembler." In this chapter, we'll be primarily discussing EDTASM. Some of our discussion will also apply to the Disk Assembler. The two assemblers are similar in that they both assemble Z-80 assembly-language code, but they are different in the ways they load the code. In this book we'll be talking about using both EDTASM and the Disk Assembler, and to keep things straight, we'll note any differences in format or technique as we go along.

EDTASM is more of a "single-user" assembler for short programs, while the Disk Assembler is normally used for more advanced work and larger programs. Chapter 13 has a large program that has been assembled by EDTASM, and we'll use that program for frequent examples in this and other chapters.

A Look At EDTASM

For openers, let's take a look at EDTASM. Some of this material might be familiar to you from using EDTASM or from reading *TRS-80 Assembly-Language Programming*. The operations in EDTASM are more straightforward than some of the advanced features of the Disk Assembler, so we'll start with the operation of EDTASM and then use that as a base for discussing the Disk Assembler.

As we know from Chapter 1, EDTASM includes both an Editor and an Assembler. The Editor has commands and subcommands similar to the Editor in Level II BASIC. Using these commands, an assembly-language source file can be initialized, modified and written out to cassette tape for subsequent assembly by the Assembler portion of EDTASM. (As an alternate approach, the source lines can be assembled directly from the buffer in memory to check for a valid assembly before writing the file out to cassette.) We won't go into the editor commands and subcommands as they're covered both in the *EDTASM User Instruction Manual* and in our previous book.

Hints and Kinks 2-1
Typical Edit Sequence

Here's a typical edit sequence to create an assembly-language source file with EDTASM:

1. Load EDTASM.
2. *I100,10 starts lines at 100 with increments of 10.
3. Enter source lines. Each line is terminated by an ENTER. Use tabs between labels, op-codes, operands, and comments.
4. Hit BREAK. This brings you back to the Editor command mode.
5. W NAME. Write file to cassette with name NAME.

See the Editor/Assembler User Instruction Manual for directions on Editor commands and subcommands.

What we **are** going to cover in this section are the subtleties of EDTASM source language **syntax** and **pseudo-ops**. Syntax (please, no bad jokes) refers to the rules of structuring assembly-language lines, while pseudo-ops are commands to the assembler in the **op-code** field of the assembly-language line. For clarity, we'll use

examples from a program assembled by EDTASM in the last section of this book.

Source Line Syntax

Figure 2-1 shows a typical section of code for the MORG program. The syntax is fairly straightforward. The source lines are **free format** in that there are no specified columns for labels, op-codes, operands, and comments. You can write source code in any convenient format as long as there's a space between each field, as long as an op-code does not start in column 1, and as long as a semicolon precedes a comment.

```

      COMMENT LINE STARTED BY SEMICOLON
      SEMICOLON PRECEDES COMMENT FIELD

01500 .
01510 : *****DEFINE MESSAGE N ROUTINE*****
01520 :
80EB 2AE985 01530 DEFINE LD HL,(CURCUR) ;GET CURRENT CURSOR
80EE 22EB85 01540 LD (LSTCUR),HL ;SAVE
80F1 CDFB83 01550 DEF005 CALL CLRCOM ;CLEAR COMMUNICATIONS AREA
8DF4 21F586 01560 LD HL,MSG5 ;DEFINE MESSAGE
8DF7 01403F 01570 LD BC,LINE13 ;LINE 13
80FA CD6D83 01580 CALL DSPMES ;DISPLAY DEFINE MESSAGE
80FD 0601 01590 LD B.1 ;1 CHARACTER
80FF CD0D84 01600 CALL INPUTS ;GET CHARACTER
8102 C27F81 01610 JP NZ,DEF050 ;GO IF GT 1
8105 D601 01620 LD B.1 ;1 CHARACTER
8107 CD9085 01630 CALL DECBIN ;CONVERT TO BINARY
810A C27F81 01640 JP NZ,DEF050 ;GO IF ERROR
810D 7D 01650 LD A,L ;MSG # NOW IN A
810E 21C987 01660 LD HL,MSG11 ;INPUT MESSAGE
8111 01803F 0167D LD BC,LINE14 ;LINE 14
8114 CD6D83 01680 CALL DSPMES ;DISPLAY MESSAGE
8117 2AE885 01690 LD HL,(LSTCUR) ;GET OLD CURSOR
811A 22E985 01700 LD (CURCUR),HL ;RESTORE
811D F5 01710 PUSH AF ;SAVE MESSAGE #
811E CDC782 01720 CALL FNDMSG ;GET ADDRESS OF MESSAGE
8121 202A 01730 JR NZ,DEF035 ;GO IF NO CURRENT MSG FOR #
      TAB (RIGHT ARROW) USED
      TO SPACE TO PRESET
      FIELDS (OPTIONAL)
01740 ; CURRENT MSG FOR # IN MBUF - MUST DELETE
8123 E5 01750 PUSH HL ;SAVE START
  
```

Figure 2-1. Assembly Source Code

Because we like neat listings, we've used the **right arrow** tab function in this and other codes to tab to the next tab position. You might also notice that when loops occur, we've indented the loop in the comments column to make them easier to follow.

A small sermon here from one who has learned the hard way: you can't use too many comments in a listing. Invariably I find myself looking at assembly-language code months later and wondering why I did a particular thing. Comments help!

The labels for assembly-language lines are optional. In most of the programs here, we've used the following general rules for labels:

1. The first label of a subroutine or main section of code is a descriptive name for the code, such as SCROLL or CLRCOM.
2. The labels in that particular section of code use the first three letters of the descriptive name plus three digits.
3. The digits in the labels are generally in ascending order. For example, INP020 follows INP010.

Of course, you can use any scheme you wish for labels. An approach like the one above does make it easier to locate code and ultimately makes coding easier.

The op-codes in the source code follow **standard Zilog mnemonics**. These are the mnemonics that Zilog, the designer of the Z-80, defined for each instruction.

The operands also follow the Zilog format for arguments. This format uses parentheses to indicate a memory address and no parentheses to indicate an immediate value; it also fixes the number of arguments for any particular instruction. Let's illustrate that first point about parentheses: If a source line appears such as LD HL,(3880H), the parentheses indicate that the contents of memory location 3880H (and 3801H) will be loaded into the HL register pair. But if a source line, (such as LD HL,3880H) appears, the lack of parentheses indicates that the **value** 3880H will be loaded into the HL register pair.

Flag Mnemonics

The mnemonics used in conditional jumps refer to flag conditions. Typical instructions might be

```
JP NZ,LOCN    ;JUMP IF NOT ZERO
JR C,LOOP     ;JUMP IF CARRY
```

The mnemonics and associated flag settings are:

<u>MNEMONIC</u>	<u>FLAG SETTING</u>
NZ (non-zero)	Z=0
Z (zero)	Z=1
NC (no carry)	C=0
C (carry)	C=1
PO (parity odd)	P/V=0
PE (parity even)	P/V=1
P (sign positive)	S=0
M (sign negative)	S=1

There are no conditional jumps for H (half-carry flag) or N (add/subtract flag). These are used internally by the Z-80 in instruction execution.

The most frequent conditional jump is on the Z flag, next the carry, next sign, next the "overflow" condition of P/V, and next the "Parity" condition of P/V.

Similarly, LD A,(HL) means that HL is used as a memory indirect pointer register and that the contents of the location pointed to by HL will be loaded into A. LD A,B, of course, means the contents of B will be loaded into A. Note that the format for Z-80 instructions in the operand field is always **destination, source**. The source operand, whether memory or register, is always last, while the result, or destination, is always first.

Constants

Now a word about numeric **constants**. As you've probably noticed, we've been using H right along as a suffix for a hexadecimal constant. Any hex value must have an H as a suffix to indicate the data is hexadecimal. If the first digit of the hexadecimal constant is A through F, legitimate hex digits, then a 0 must be added before the hex digit. It's easy to see why this is so since the Assembler must be able to differentiate between constants and labels.

If a constant has no suffix, it's assumed to be a decimal constant. A suffix of D signifies a decimal constant (so there's no point in using it!), and a suffix of O signifies an octal constant (so there's probably no need for this suffix either, since you'll probably never require octal constants).

If a character is bracketed by single quotation marks, it's interpreted as an ASCII character. To load the B register with an ASCII A, for example, we'd have
LD B, 'A' ; LOAD A INTO B REGISTER.

We'll discuss **expressions** involving constants and symbols after we look into the use of **pseudo-operations**.


Pseudo-Operations

Most assembly-language source lines are **generative** types of lines. A mnemonic representing a Z-80 op-code generates the corresponding machine code for that instruction at that point in the assembly. However, there are a number of commands to the assembler, called pseudo-operations or "pseudo-ops," that don't generate instructions. Instead, they inform the Assembler of certain actions to be taken, or they create data values.

The first of these is the **Origin**, or **ORG**, pseudo-op. The Origin informs the Assembler that the following code is to be assembled for a particular memory location. For an example of this, look at Figure 2-2, where **MORG** has been assembled to run at 8000H. When the Assembler encounters the **ORG** statement, it will set an internal

assembly location counter to the value of the operand in the ORG statement. This assembly location counter will be adjusted by the length of each instruction as an instruction is assembled. Unless the code is **relocatable**, code produced for one Origin will not run anywhere else in memory, since there are addresses in many instructions which are **absolute** addresses. (We'll discuss **relocatability** shortly.)

SETS ASSEMBLY LOCATION COUNTER TO 8000H



```

8000      00100      ORG      8000H
00110      ;MORG-0820
00120      ;*****MORSE CODE GENERATOR PROGRAM*****
00130      ;
00140      ;
00150      ;*****SYSTEM EQUATES*****
00160      ;
3C00      00170 SCREEN EQU      3C00H      ;START OF VIDEO DISPLAY
3C40      00180 LINE1 EQU      SCREEN+64    ;SECOND LINE
3EC0      00190 LINE11 EQU     SCREEN+704    ;TWELFTH LINE
3F00      00200 LINE12 EQU     SCREEN+768    ;THIRTEENTH LINE
3F40      00210 LINE13 EQU     SCREEN+832    ;FOURTEENTH LINE
3F80      00220 LINE14 EQU     SCREEN+896    ;FIFTEENTH LINE
3FC0      00230 LINE15 LQU     SCREEN+960    ;SIXTEENTH LINE
0002      00240 ENTER EQU      2            ;ENTER CHARACTER
0001      00250 CLEAR EQU      1            ;CLEAR CHARACTER
0064      00260 DBDEL EQU      100          ;DEBOUNCE DELAY IN MS
0065      00270 DBDEL1 EQU     DBDEL+1      ;DEBOUNCE DELAY+1 MS
000A      00280 MLDEL EQU      10          ;MAIN LOOP DELAY IN 1/10 MS
03C0      00290 SPEEDF EQU     960          ;FINAGLE FACTOR FOR SPEED
          00300      ;
          00310      ;*****MORSE EXECUTIVE*****
          00320      ;
8000 F3    00330 START  EI              ;DISABLE INTERRUPTS
8001 A315994 00340      LD      SP,TOPS    ;SET STACK POINTER
8004 11EA88 00350      LD      DE,HBUF    ;MESSAGE BUFFER ADDRESS
8007 010B0A 00360      LD      BC,2571    ;2571 BYTES

FIRST INSTRUCTION
ASSEMBLED AT
LOCATION 8000H

```

Figure 2-2. ORG Use

The END pseudo-op marks the end of the source file and is self-descriptive. If the END has an operand, use it as the starting point in the program when the program is loaded as in END START.

There are three pseudo-ops that generate data in EDTASM. The first of these is DEFB, or DEFIne Byte, which generates a single 8-bit value. The next is DEFW, or DEFIne Word, which generates a 16-bit value. The last is DEFM, or

DEFine Message, which generates a string of ASCII characters that usually represents a message used in the program. Each of these three pseudo-ops can have a label, if desired. Examples of each are shown in Figure 2-3.

```

D96DD ;*****WDRKING STDRAGE*****
0961D ;
85DC 0000 09620 TMP1 DEFW 0 ;TEMPORARY STDRAGE
850E 0D D963D PRINTF DEFB D ;PRINTER FLAG:D=OFF,1=DN
85DF 00 09640 LPFTF OEFB 0 ;LP 1ST TIME FLAG:0=1ST TIME
85ED 0D D965D CHARCT DEFB 0 ;LP CHARACTER COUNTER
85E1 D204 D9660 SEED DEFW 1234 ;DEFAULT SEED
85E3 2E16 0967D OEFW 5678
85E5 9DD1 09680 DOTD DEFW 400 ;OOT DN TIME (3 WPM DEFAULT)
85E7 BDD4 D969D DASHD DEFW 12DD ;DASH DN TIME (3 WPM OEFLT)
85E9 DD3C D97D0 CURCUR DEFW 3CDDH ;CURRENT CURSDR PDSITIDN
85EB DD3C D9710 LSTCUR DEFW 3CDDH ;LAST CURSDR PDSITIDN
85ED DDDD D9720 TSLC DEFW D ;TIME IN MS SINCE LAST CHAR
85EF 2D D973D LASTR OEFB ' ' ;LAST RANDOM CHARACTER SENT
85FD F592 D9740 IBUFL DEFW IBUF ;PINTER TO LAST IBUF SLDT
85F2 F592 D975D IBUFN DEFW IBUF ;POINTER TD NEXT IBUF SLDT
      0976D ;
      D977D ;*****SYSTEM MESSAGES*****
      D978D ;
85F4 2D D979D MSG1 OEFM ' ***MDRG***
      2D 2D 2D 2D 2D 2D 2D 2D
      2D 2D 2D 2D 2D 2D 2D 2D
      2D 2D 2D 2D 2D 2D 2D 2D
      2D 2D 2D 2A 2A 2A 4D 4F
      52 47 2A 2A 2A 2D 2D 2D
      2D 2D 2D 2D 2D 2D 2D 2D
      2D 2D 2D 2D 2D 2D 2D 2D
      2D 2D 2D 2D 2D 2D 2D 2D
8634 43 0980D DEFM 'CHAR=SEND CHARACTER SHIFT D-9=SEND MSG N'
      48 41 52 3D 53 45 4E 44
      2D 43 48 41 52 41 43 54
      45 52 2D 2D 53 48 49 46
      54 2D 3D 2D 39 3D 53 45
      4E 44 2D 4D 53 47 2D 4E
865D 2D 0981D DEFM ' SHIFT R=SEND RANDOM SHIFT D=DEFINE MS'
      2D 53 48 49 46 54 2D 52
      3D 53 45 4E 44 2D 52 41
      4E 44 4F 4D 2D 2D 53 48
      49 46 54 2D 44 3D 44 45
      46 49 4E 45 2D 4D 53
8685 47 D982D DEFM 'G SHIFT S=DEFINE SPEED SHIFT P,N=PRINT'
      2D 2D 2D 53 48 49 46 54
      2D 53 3D 44 45 46 49 4E
      45 2D 53 5D 45 45 44 2D
      2D 53 48 49 46 54 2D 5D
      2C 4E 3D 5D 52 49 4E 54
86AE 2D D983D DEFM ' DR NO'
      4F 52 2D 4E 4F

```

Figure 2-3. DEFB, DEFW, DEFM Use

The DEFW generates 16-bits of data in standard Z-80 word format. In this format, the **least significant byte** of the 16 bits is stored in the first byte, and the **most significant byte** of the 16 bits is stored in the second byte. If you look at the machine code produced in Figure 2-3, you'll see the hexadecimal data generated in CURCUR DEFW 3C00H, for example, is arranged as 003C.

Hints and Kinks 2-3
Sixteen-Bit Data Format

Many programmers get confused over 16-bit data format. When a 16-bit piece of data is stored in memory, it is always stored least significant byte followed by most significant byte. The code

```
LD      (LOCN),HL      ;STORE HL
LD      (LOCN+2),DE     ;STORE DE
LD      (LOCN+4),BC     ;STORE BC
```

would store data in the LOCN area as follows:

LOCN + 0	L
+ 1	H
+ 2	E
+ 3	D
+ 4	C
+ 5	B

Data in the stack (say, a PUSH HL) is stored the same way.

When data is retrieved (LD DE,(LOCN+2) or POP BC) it's put back into the registers in identical fashion. (Whew! What if the designers had chosen to retrieve it in opposite fashion.... Thank goodness for cool heads in Silicon Valley...).

The **DEFM** simply generates a one-byte ASCII character for every character in the string. The string is started and terminated by a single quotation mark (').

The **DEFS** (DEFine Storage) pseudo-op reserves a number of bytes at the current assembly location. If, for example, you want to define a table that would be filled with values during program execution, you might have the following code:

```
TABLE DEFS 100      ;100-BYTE TABLE
```

The assembler would then increase the assembly location counter by 100, bypassing 100 bytes. Note that the machine code bytes produced for a DEFS consist of garbage, or unknown, data. A DEFS does not fill the vacant space with zeroes, all ones, or anything; it just leaves whatever is there to begin with.

The last pseudo-op is EQU, or EQUate. To understand this pseudo-op, you need to understand the processing that the Assembler goes through. The Assembler makes two (or more) **passes** through the source file. After the first pass, it has assembled a **symbol table** of all labels with a corresponding numeric value for each symbol. (This is the information displayed at the end of the program listing.)

In most cases, each symbol has a value that represents the assembler location counter value, which is essentially the location at which the instruction or data for that symbol will reside. The EQU pseudo-op, however, can force the Assembler to equate a label either with a numeric value or another label. This is the case for CDTAB as shown in Figure 2-4. CDTAB is equated to CTAB. In other words, the value associated with CTAB — the location of 8812H — will also be used for symbol CDTAB. In this case, we made the association because the CDTAB contained the same data as the first 44 locations of CTAB.

```

10080 :*****CTAB CHARACTER TABLE*****
10090 :*   TABLE OF CHARACTERS TO BE SENT IN RANDOM MOOE.   *
10100 :*   OISTRIBUTION DOES NOT CORRESPOND TO THAT IN NOR-  *
10110 :*   MAL TEXT. SPACE CHARACTER NOMINALLY EVERY 5TH    *
10120 :*   CHARACTER.                                         *
10130 :
8812 41 10140 CTAB      OEFH      'ABCOEFGHIJKLMNOPQRSTUVWXYZ0123456789..?/- =,'
      42 43 44 45 46 47 48 49
      4A 4B 4C 4D 4E 4F 50 51
      52 53 54 55 56 57 58 59
      5A 30 31 32 33 34 35 36
      37 38 39 2E 2C 3F 2F 2D
      20 3D 3B
883E 30 10150          OEFH      '0123456789..?/- =, ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      31 32 33 34 35 36 37 38
      39 2E 2C 3F 2F 20 20 30
      3B 41 42 43 44 45 46 47
      48 49 4A 4B 4C 4D 4E 4F
      50 51 52 53 54 55 56 57
      58 59 5A
886A 20 10160          OEFH      '
      20 20 20 20 20 20 20 20
      20 20 20 20 20 20 20 20
      20
887C 41 10170          OEFH      'ABCOEFGHIJKLMNOPRSTUVY'
      42 43 44 45 46 47 48 49
      4A 4B 4C 4D 4E 4F 50 52
      53 54 55 56 59

```

← THESE BYTES ARE CDTAB!

```

10180 ;
10190 ;*****CDTAB CODE TABLE*****
10200 ;*   TABLE OF VALID ASCII CHARACTERS TO BE TRANSMITTED.*
10210 ;*   INDEX TO CHARACTER USED TO OBTAIN TIMING CODES   *
10220 ;*   FROM TTAB.                                         *
10230 ;
8812 10240 CDTAB    EQU      CTAB          ;SAME DATA
002C 10250 CDTABS  EQU      44           ;SIZE OF DATA
10260 ;

```

Figure 2-4. EQU Use Example 1

As figure 2-4 also shows, you can use the EQU pseudo-op for CDTABS. CDTABS is equated to 44. Every time CDTABS is referenced, as in LD B,CDTABS, the value loaded for CDTABS will be 44. LD B,CDTABS would therefore become equivalent to LD B,44. Equates are used in this fashion so an easily-remembered symbolic name can be given to constants such as size of tables, addresses of I/O devices, etc.

Another use of EQU is shown in Figure 2-5. Here MBUF (Message Buffer), has been equated to \$. \$ is a special symbol that represents the current value of the assembler location counter. After the MBUF equate, MBUF would be stored in the symbol table as the value 88EAH. The next line, ENDM EQU \$+2571, equates the label ENDM to \$+2571. Since the assembler location counter did not change from the previous equate (no instructions or data were generated), ENDM is equated to 88EAH+2571 or 92F5H. This little trick is synonymous to:

```

MBUF EQU $

      DEFS 2571

ENDM EQU $

```

and is quite commonly used.

```

10780 ;*****MESSAGE BUFFER*****
10790 ;*   ARBITRARILY SET AT 2560 BYTES (256 BYTES PER MSG   *
10800 ;*   PLUS MSG# PLUS 1 TERMINATOR).                       *
10810 ;
88EA 10820 MBUF    EQU      $
92F5 10830 ENDM    EQU      $+2571
10840 ;

```

Figure 2-5. EQU Use Example 2

Operators and Expressions

EDTASM allows a limited number of operations involving addition, subtraction, ANDing, and shifting. These operations are performed by the **operators** +, -, &, and <. Addition of constants are shown in Figure 2-6, along with one subtraction example for the size of FTAB. Expressions may also include a mixture of numeric data and labels in any combination.

```

                                01130 ;
                                01140 ; FUNCTION TABLE
                                01150 ;
80BE C4                        01160 FTAB    OEFB    'D'+80H        ;OEFINE MESSAGE
80BF 03                        01170        OEFB    'S'+80H        ;OEFINE SPEED
80C0 02                        01180        OEFB    'R'+80H        ;TRANSHIT RANOOM
80C1 B0                        01190        OEFB    'O'+80H        ;TRANSHIT MESSAGE 0
80C2 B1                        01200        OEFB    '1'+80H        ;
80C3 B2                        01210        OEFB    '2'+80H        ;
80C4 B3                        01220        OEFB    '3'+80H        ;
80C5 B4                        01230        OEFB    '4'+80H        ;
80C6 B5                        01240        OEFB    '5'+80H        ;
80C7 B6                        01250        OEFB    '6'+80H        ;
80C8 B7                        01260        OEFB    '7'+80H        ;
80C9 B8                        01270        OEFB    '8'+80H        ;
80CA B9                        01280        DEFB    '9'+80H        ;
80CB D0                        01290        DEFB    'P'+80H        ;SET PRINT
80CC CE                        01300        DEFB    'N'+80H        ;RESET PRINT
000F                          01310 FTABS    EQU    $-FTAB        ;SIZE OF FUNCTION TABLE
0000                          01320        ENO
00000 TOTAL ERRORS

```

Figure 2-6. Operators and Expressions

Using EDTASM to Edit and Assemble Programs

EDTASM is geared to editing and assembling one **module** of assembly-language code. After you have specified, flow-charted or analyzed, and coded a program on paper, use the Editor in EDTASM to enter the complete program code. Although you have probably divided the program into a number of different functional modules, include the modules as one large, whole assembly-language source file.

Typical Assembly Sequence

With the source file in memory (either from cassette or from keyboard entry), a typical assembly sequence might be:

1. *A/NO/WE Assemble to screen with no object; wait on errors.
2. Go back to Editor to correct any errors. When there are no errors, go to 3.
3. *A/NO/LP Assemble to line printer with no object.
4. Take the listing and do a comprehensive desk check (Coffee or other stimulants allowed.)
5. Repeat steps 1 through 4 as often as required.
6. *A NAME/LP Assemble final assembly to line printer, object to cassette with name "NAME".
7. Debug.

The resulting source file on cassette is then assembled as one large assembly. MORG, the Morse Code Program of Chapter 13, is a large program that approaches the limits of memory capacity. Because all of the source lines must be held in memory at one time, in addition to the symbol table, there's a practical limit to the size of the program that can be assembled under EDTASM. This limit depends upon the memory size of the system, number of characters in the source file, and number of labels used. We'll see in the next chapter how you can use the Disk Assembler to overcome some of the limitations by using a different approach to assembling and loading programs, but let's first get a clear understanding of EDTASM operation.

As we can see from looking at MORG, the entire program is assembled in one swell foop. The ORG statement specifies the starting address of the program, and all instructions are referenced to the assembler location counter. The assembler location counter is initially set by ORG and incremented as each instruction or pseudo-op is generated. EDTASM produces **object code** for the program as a cassette file. The object code is very similar to the machine code shown on the listing, but contains some additional data to hold the file name, origin, number of bytes per record, checksum, etc.

You can load the resulting object code by a SYSTEM command while in Level II BASIC monitor mode. The SYSTEM command enables you to load the object file created under EDTASM and then to transfer control to the starting address of the program (specified by the operand used with the END pseudo-op).

The object code produced by EDTASM generally can be loaded and executed only at one point in memory, the area at which it was ORiGined. If you load the object code at another area (by some devious means), it won't execute properly. Let's see why this is so.

Relocatability

Instructions in the Z-80 instruction set are generally **relocatable** or **non-relocatable**. Relocatable instructions will execute properly anywhere in memory; non-relocatable instructions contain absolute references and will execute only in the area for which they were assembled.

Why All the Interest in Relocatability?

Much Z-80 literature talks about the relocatable instructions or code and touts the advantages. Are there many advantages in the relocatable Z-80 instructions?

Not really. Any large piece of code is probably not relocatable because it will have to contain JPs and CALLs – unless some pretty clever coding is done. About the only reason for relocatability on the TRS-80 is to allow short code segments to be embedded in BASIC programs. This merges BASIC and assembly-language code and allows the programmer to use assembly-language code to speed up his time critical processing.

Occasionally someone will try to take an existing program that is available in machine language only (no listing) and attempt to relocate it to run elsewhere. This is possible, but very tedious. Try it only on a favorable bio-rhythm day!

Figure 2-7 shows typical code for the MORG program produced by EDTASM. Let's take a look at the instructions to see which are relocatable, which are not, and why.

NOT RELOCATABLE						
8197	CDFB83	02340	SPE005	CALL	CLRCON	;CLEAR COMMUNICATION AREA
819A	21CB86	02350		LD	HL,MSG4	;SPEED MESSAGE
8190	01403F	02360		LD	BC,LINE13	;LINE 13
81A0	CD6D83	02370		CALL	OSPMES	;DISPLAY SPEED MESSAGE
81A3	D602	02380		LD	B,2	;2 CHARS
81A5	C00684	02390		CALL	INPUTS	;GET CHARACTER STRING
81A8	2035	02400		JR	NZ,SPE020	;GO IF GT 2 CHARACTERS
81AA	CD9085	02410		CALL	DECBIN	;CONVERT TO BINARY
81AD	2030	02420		JR	NZ,SPE020	;GO IF ERROR
81AF	70	02430		LD	A,L	;GET SPEED 0-99
81B0	FE03	02440		CP	3	;TEST FOR 3 WPM
81B2	FA0F81	02450		JP	M,SPE020	;GO IF LT 3 WPM
81B5	FE30	02460		CP	61	;TEST FDR 60 WPM
81B7	F20F81	02470		JP	P,SPED20	;GO IF GT 60 WPM
81BA	21CD03	02480		LD	HL,SPE0F	;1200/WPM=DOTO TIME
81B0	4F	02490		LD	C,A	;WPM LS BYTE
81BE	0600	02500		LD	B,0	;NOW IN BC
81C0	11FFFF	02510		LD	DE,-1	;QUOTIENT
RELOCATABLE						

Figure 2-7. Relocatable Code

The CP 61 found 12 lines after SPE005 has machine-language code of FE3D. By reference to Appendix II, we can see that the first byte of this instruction is the op-code and that the second byte is the data value of 61 (3DH). This instruction would assemble exactly the same way at any spot in memory, because the op-code is fixed and the data value would have to be the same.

The instruction CALL CLRCOM, however, is a different beastie. CLRCOM is a subroutine that appears somewhat later in the program. If you'll look again at the formats in Appendix II, you'll see that a CALL consists of three bytes. The first of these is an op-code of CDH. The second and third bytes are an **absolute memory address** that hold the call address with the bytes in reverse order. The address in this case is that of CLRCOM, which is at 83FBH. Would this instruction execute properly if MORG were moved to another area of memory? Obviously not, as the CLRCOM subroutine would also be relocated, and its address would be other than 83FBH. A similar situation would exist for instructions such as JP SPE020, LD HL,MSG4, and others that contain absolute addresses.

You might be wondering if it's possible to write programs in pure **relocatable code**. As a matter of fact, you can write **relocatable code** in relatively short routines that contain no absolute addresses and use JR type jumps for conditional and unconditional jumps. We'll explore some of the techniques in Chapter 5. Longer programs require you to do something such as reassembling your program by EDTASM with a new Origin. Another way is to use the facilities of the Disk Assembler. We'll look into this last method in the next chapter.

Chapter Three

Assemblers and Assembling — Disk Assembler

In this chapter we'll talk about the Radio Shack Disk Assembler. You should use this material only as a supplement to the *Disk Assembler User Instruction Manual*, since an adequate discussion of the Assembler would require more than just a single chapter. A good example for some of the techniques associated with the Assembler is included in Chapter 14 where we illustrate an entire program made up of modules assembled and loaded by the Disk Assembler and Loader. We'll be using further examples in the text of Disk Assembler code and EDTASM code in the remaining chapters of this book.

Comparison Between EDTASM and the Disk Assembler

Although EDTASM is cassette-based and the Disk Assembler must use files from disk, the two systems share more similarities than differences. Both edit assembly-language source files and have approximately the same line-oriented type of Editor with very similar commands and subcommands. Both assemble the source files and look for identical formats as far as Z-80 mnemonics and instruction syntax. Both use pseudo-ops for origin; pseudo-ops for definition of bytes, words, text, and storage; and a pseudo-op for equates. Many EDTASM source files can be converted to Disk Assembler use quite easily with a minimum of editing changes, primarily by adding a colon after each label and changing some of the data definitions.

Hints and Kinks 3-1
Format Differences between
EDTASM and Disk Assembler

There are several format and pseudo-op differences between the two assemblers. The most obvious is the use of a colon after labels on most source lines (not on an EQU).

The pseudo-ops for data definition are also somewhat different. Although DEFB, DEFW, DEFS and DEFM may be used, the Disk Assembler also uses DB in place of DEFB, DW in place of DEFW, DS in place of DEFS, and DC in place of DEFM. DB 'string' can also be used as a DEFS.

Multiple arguments can be used for the data definition pseudo-ops as in "DEFB 2,3,5".

The Disk Assembler also requires at least a 95 character wide line for assembly listing.

Can you convert EDTASM files on cassette to Disk Assembler source files? Conceivably, a short assembly-language conversion program might be used that would convert the format differences above (and others). It would make a nice exercise . . .

The simple uses of the Disk Assembler look very similar to EDTASM. However, when all the capabilities of the Disk Assembler are utilized, there are some major advantages to the Disk Assembler not found in EDTASM. In approximate order of importance, these are:

1. The Disk Assembler uses disk files for source, object, and listing.
2. The Disk Assembler produces object-file output in the form of **relocatable object modules** which are loaded by a special Loader.
3. The Disk Assembler has a macro capability to generate in-line macro code.
4. The Disk Assembler has a number of pseudo-ops relating to program **sections** and conditional assembly.

5. The Disk Assembler has a number of pseudo-ops relating to listing format.

We'll discuss each of these points in the following text, and give a number of examples to illustrate each concept.

Disk Files for the Disk Assembler

The Disk Assembler operates on assembly-language source files on disk created by the Editor portion of the software package. These source files are processed by the Assembler and the resulting object files are output to disk. The object files can be **loaded** into memory by the Loader portion of the package, and a **CMD** (CoMmanD) file can then be dumped to disk.

The **CMD** file, is a **core-image** file of the machine-language code that makes up the complete assembly-language program. It can be loaded and executed by simply typing in the name of the **CMD** file after the TRSDOS "DOS READY" prompt.

In addition to the source, object, and **CMD** files created by the Editor and Assembler, a **listing file** can be created for a hardcopy listing. The listing file can then be "PRINTed" on the TRS-80 system line printer.

Using the Editor for Source Files

The assembly-language source file is similar to other files, but there are significant differences. The assembly-language source file created by the Editor is almost an ASCII file. However, there is some non-ASCII coding for the line numbers at the start of each line. The Editor commands are similar to the Editor commands in Level II BASIC or EDTASM, with some differences. The symbol "." is used to refer to the current line and the symbol "*" refers to the last line of the **edit buffer**, just as in the other editors. **The first line of the buffer, however, is signified by the symbol, "⬆" (up arrow), rather than "#".**

Ranges of lines may be specified, just as in the other editors. For example, to delete lines 200 through 300, give

the Editor command D200:300. A new type of range specification, however, deletes a specified line through the next *n* lines. D200!3, for instance, would delete line 200 and the next 3 lines following line 200.

The Editor commands — I (Insert), D (Delete), R (Replace), P (Print), and N (Number) — all operate the same as in the other editors. To edit a specific line, though, the command A (Alter), rather than E is given. See Table 3-1.

Table 3-1
Disk Editor/Assembler Edit Commands

Command	Action	Typical Example*
A	Alter	A100 starts edit (Alter) of line 100
B	Begin	B / 2 moves to beginning of page 2
D	Delete line	D100:200 deletes lines 100-200
E	Exit Editor	E NAME/MAC exits Editor, writes file NAME
F	Find	F100:200\$TWINE\$ finds string "TWINE"
I	Insert	I100,10 inserts lines from 100, increment 10
K	Kill	K / 2 kills page mark at page 2
L	List	L100:* prints lines from 100 through end
M	Mark	M100 inserts page mark after line 100
N	Number	N100,10 renumbers lines from 100, increment of 10
P	Print	P100:200 displays lines 100-200
Q	Quit	Q quits Editor without disk write
R	Replace	R100 replaces line 100
S	Substitute	S100:200\$MOD II\$MOD III\$ substitutes "MOD III"
W	Write to disk	W NAME/MAC writes text as disk file "NAME/MAC"
X	Extend	X100:200 enters extend mode for lines 100-200

*Consult Disk Editor/Assembler manual for the many variations.
\$ = BREAK

Editor **subcommands**, commands that operate within a specified line, are very similar to the other editors. You'll discover some new subcommands in this mode, though. See Table 3-2 for a complete list.

Table 3-2
Disk Editor/Assembler Edit Subcommands

Subcommand	Action	"Usual" Form *
A	Prints remainder, enters changes, concludes editing	A
B(lanks)	Inserts spaces	B or 1B
C(hange)	Replaces characters	C <ch> or 1C <text>
D(elete)	Deletes characters	D or 1D
E(nter)	Enters changes, concludes editing	E
F(ind)	Finds text	F\$<text>\$
G	Inserts characters	G<ch>
H(acks)	Delete remainder and enter Insert mode	H or H<text>\$
I(nsert)	Inserts text	I or I<text>\$
K(ill)	Kills characters up to <ch>	K<ch>
L(ine)	Prints line, positions cursor to beginning	L
N(ot)	Restores line, moves to next line	N
O(bliterate)	Deletes all characters up to <text>	O<text>\$
P(osition)	Prints remainder of line, cursor static	P
Q(uit)	Exits edit mode and restores original line	Q
R(eplace)	Replaces characters with <text>	1R<text>\$
S(earch)	Finds character <ch>	S<ch>
T(runcate)	Deletes remainder of line and concludes edit	T
W(ord)	Moves the cursor to beginning of next word	W
(e)X(tend)	Print remainder of line, go into Insert mode	X
Z(aps)	Deletes word	Z
←	Delete character	←
ENTER	Prints remainder, enters changes, concludes edit	ENTER
SHIFT ←	Restores original line, repositions cursor to beginning	SHIFT ←
SPACE	Spaces over character	SPACE
→	Moves cursor to line end	→

i = number

\$ = BREAK or ENTER

<ch> = character

<text> = text string

* Consult Disk Editor/Assembler manual for the many variations

Here's the procedure for creating a new source file:

1. Load the Editor by entering EDIT after the TRSDOS READY prompt. The Editor will be loaded as a CMD file and will start execution.
2. The Editor will ask for the file name by displaying FILE:.
3. Enter a TRSDOS file name with optional extension, password, and drive number; **follow that with the BREAK key**. Since the Assembler works with the extension /MAC, this would be the best choice for the extension. You can also disregard passwords, unless you're really paranoid about your source files! Some examples of source file names might be MTALP/MAC:1 or HTPM/MAC.
4. You'll now be at the command level, as indicated by an asterisk (*), and can start creation of source lines by insert commands such as I100.

To modify an existing source file, follow a similar sequence to the one above, only type an ENTER rather than a BREAK after the file name.

During the source file editing, you can P(rint) the assembly source lines on the screen, or get a hardcopy on your system line printer by the L(ist) command. During the editing session, you will notice some **disk activity** taking place. The Editor will be reading in **pages** for the source file as required.

When you have the source file the way you'd like it, you can write a new source file to the disk by giving the E(nd) command and the name initially used. If this is modification of an existing file, the form of the command is E NAME/MAC, where the NAME of the source file must be different than the one originally read in.

This last point means that when you edit an existing file, you must first read in the old file, perform editing on the

old file, create a new file at the end of the edit session, KILL the old file name under TRSDOS, and then RENAME the new file name to the old name to get back to square one. Though somewhat tedious, this goes quite rapidly and does offer some automatic protection against **clobbering** an old file before its time.

—Hints and Kinks 3-2 —

Typical Edit, Rename Sequence

Here's a typical Edit and rename sequence:

DOS READY	
EDIT	Load Editor
FILE: EQUATE/MAC	Read in old file
(title, copyright)	
*A100	Edit line 100
00100 TITLE EQUATE	
*E EQUATE1/MAC	End edit, write out
DOS READY	
KILL EQUATE/MAC	Kill old file
DOS READY	
RENAME EQUATE 1/MAC TO	Rename new to old
EQUATE/MAC	name
DOS READY	

Relocatable Object File Assembly

The Assembler will read in the source file created by the Editor, assemble it, and produce a **relocatable object file**, a **listing file**, or both on disk. We need to talk about the relocatable object file, since the philosophy here is quite different from the EDTASM approach.

Figure 3-1 shows a typical assembly listing for the Disk Assembler. (This is a module of the Tic-Tac-Toe Artificial Intelligence Program of Chapter 14.) The appearance of this assembly is very similar to what EDTASM would produce. The **source line image** and the edit line number portion are almost identical.

Figure 3-1. Typical Disk Assembler Listing

```

00100      TITLE MAIN1
00110      ENTRY ARTIP
00120      EXT SCREEN, SCARRY, LINE13
00130      EXT NUMBER, ANALAR, ROTATE, ARRXLA, RAND, INPUT
00140      EXT FILLCH, OSPMES, DRAWL, MSGOUT, DELAY, BINBAS
00150      EXT SCRNO5, HISTUP, MEMORY, BASBIN
00160      EXT GRIOTB, NOVETB, HISTB, ARRAY1, ARRAY2, ROTTAB
00170      EXT ANALTB, RINDT, RINDTR, ARRAY1, ARRAY2, ROTTAB
00180      EXT MOVENO, FRSTF, NHTHIS, PTABLE, RINDW, NOPTE
00190      EXT NOX, NOO, NOSPF, ROTETR
00200      EXT LASTL, MSG10, MSG11, STACK
00210      EXT MSG1, MSG2, MSG3, MSG4, MSG5, MSG6, MSG7, MSG8, MSG9
00220      ;*****
00230      ;***** TIC-TAC-TOE LEARNING PROGRAM *****
00240      ;*****
00250      ;*****
00260      ;*****
00270      ;*****
00280      ;*****
00290      ;*****
00300      ;*****
00310      ;*****
00320      ;*****
00330      ;*****
00340      ;*****
00350      ;*****
00360      ;*****
00370      ;*****
00380      ;*****
00390      ;*****
00400      ;*****
00410      ;*****
00420      ;*****
00430      ;*****
00440      ;*****
00450      ;*****
00460      ;*****
00470      ;*****
00480      ;*****
00490      ;*****
00500      ;*****
00510      ;*****
00520      ;*****
00530      ;*****
00540      ;*****
00550      ;*****

0000'      31 0000*
0003'      3E 80
0005'      11 0000*
0008'      01 0340
000B'      CD 0000*
000E'      21 0000*
0011'      01 0000*
0014'      CD 0000*

0017'      DD 21 0000*
001B'      D0 7E 00
001E'      FE FF
0020'      28 16
0022'      D0 4E 01
0025'      D0 46 02
0028'      DD 6E 03
002B'      DD 66 04
002E'      CD 0000*
0031'      01 0005
0034'      D0 09
0036'      18 E3

0038'      AF
0039'      11 0000*
003C'      01 0008
003F'      CD 0000*
0042'      32 0000*

00100      TITLE MAIN1
00110      ENTRY ARTIP
00120      EXT SCREEN, SCARRY, LINE13
00130      EXT NUMBER, ANALAR, ROTATE, ARRXLA, RAND, INPUT
00140      EXT FILLCH, OSPMES, DRAWL, MSGOUT, DELAY, BINBAS
00150      EXT SCRNO5, HISTUP, MEMORY, BASBIN
00160      EXT GRIOTB, NOVETB, HISTB, ARRAY1, ARRAY2, ROTTAB
00170      EXT ANALTB, RINDT, RINDTR, ARRAY1, ARRAY2, ROTTAB
00180      EXT MOVENO, FRSTF, NHTHIS, PTABLE, RINDW, NOPTE
00190      EXT NOX, NOO, NOSPF, ROTETR
00200      EXT LASTL, MSG10, MSG11, STACK
00210      EXT MSG1, MSG2, MSG3, MSG4, MSG5, MSG6, MSG7, MSG8, MSG9
00220      ;*****
00230      ;***** TIC-TAC-TOE LEARNING PROGRAM *****
00240      ;*****
00250      ;*****
00260      ;*****
00270      ;*****
00280      ;*****
00290      ;*****
00300      ;*****
00310      ;*****
00320      ;*****
00330      ;*****
00340      ;*****
00350      ;*****
00360      ;*****
00370      ;*****
00380      ;*****
00390      ;*****
00400      ;*****
00410      ;*****
00420      ;*****
00430      ;*****
00440      ;*****
00450      ;*****
00460      ;*****
00470      ;*****
00480      ;*****
00490      ;*****
00500      ;*****
00510      ;*****
00520      ;*****
00530      ;*****
00540      ;*****
00550      ;*****

ARTIP:      LD SP, STACK
            LD A, 60H
            LD DE, SCREEN
            LD BC, 832
            CALL FILLCH
            LD HL, MSG1
            LD BC, LINE13
            CALL DSPMES
            ORAW GRID HERE
            LD IX, GRIDTB
            LD ART005: L0 A, (IX)
            CP ORFH
            JR Z, ART008
            L0 C, (IX+1)
            L0 B, (IX+2)
            L0 L, (IX+3)
            L0 H, (IX+4)
            CALL DRAWL
            LD BC, 5
            ADD IX, BC
            JR ART005
            ; ZERO MOVE TABLE
            ART008: XOR A
            LD DE, MOVETB
            LD BC, 8
            CALL FILLCH
            LD L0 (MOVENO), A
            ; TEST FOR FIRST TIME ACTIONS

```

Looking at the machine language code produced by the Assembler, we see that many of the instructions look familiar. `ADD IX,BC`, for example, generates two bytes of `DD 09`, just as it would in EDTASM. If you look at some of the `CALLs`, however, you'll notice that the address of the `CALL` is zeroes. Also, the address for `JP` instructions (not shown) appear to be low-valued, such as `0029`. Note also that 16-bit values are printed in "normal" order on the listing; in memory they are still least significant byte followed by most significant byte. What's the logic behind these differences?

In a relocatable object module containing the machine-language code for this assembly, the machine-language code is referenced to the **start** of the module. The location column on the listing starts at location 0' (the prime indicates that the code is relocatable). Jumps within the module generate instructions that contain addresses that are "displacement" addresses from the start of the module.

The **Loader** for the Disk Assembler handles the task of loading in a number of these relocatable object modules and of filling in the proper addresses for **non-relocatable** instructions such as `JPs` and `CALLs`.

Hints and Kinks 3-3

Mode Indicators

The character at the end of the location value is the mode indicator as follows:

'	Code relative
"	Data relative
!	Common relative
(space)	Absolute
*	External

These characters are also used after two-byte values.

Another thing that the Loader does is to **link** together locations from module to module. Obviously there has to be some way for modules to communicate with one another. For example, a portion of **main** code, such as is shown in Figure 3-1, might have to call a subroutine represented by another module.

The loader does this by linking together **EXT(ernals)** and **ENTRYs** from module to module. The **ENTRY** operand is a location within one module that might be referenced by code in another module, as for a **CALL** or **JP**. The **EXT** operand is a label that is external to the module but is referenced inside the module.

Look again at Figure 3-1. At location 14' a **CALL** is made to a subroutine **DSPMES**. This subroutine is external to this module and declared as an **EXT** in the pseudo-ops at the start of the program. Location **ARTIP**, however, is the main entry point and is declared as an **ENTRY**. Note that the **CALL** to **DSPMES** results in machine code of **CD 0000***. The address will be filled in by the loader at **load time**.

A complete assembly-language program is created by **link loading** a group of relocatable modules, such as the one related to the listing in Figure 3-1. The loader compiles a table of all the **EXT** and **ENTRY** points, somewhat analogous to the Assembler symbol table, and eventually goes through a "threaded list" to fill in the addresses for every externally referenced (or **global**) label. Labels within the module (**local** labels) are, of course, no problem.

There are several major advantages to this type of assemble and load scheme: (1) we are no longer limited by memory size in constructing large assembly-language packages; (2) we don't have to worry about relocation as the loader takes care of that automatically; and (3) the assembly-language design job can be more structured by dividing the design into a number of separate modules (an especially valuable feature when more than one programmer is working on a project).

There are a few minor disadvantages. For one thing, the entire edit, assemble, load process is less interactive and more time consuming. It does take somewhat more time to generate a new assembly-language package as the entire load process has to be repeated, even if only one line has been changed. However, the extra time spent is a small price to pay for the increased flexibility of the Assembler.

Macro Capability

Another feature the Disk Assembler has over EDTASM is the ability to use **macros**. Macros are essentially **in-line** subroutines generated at **assembly-time**. For example, suppose that we have six instructions that appear a dozen times in an assembly-language program:

```
ADD HL,HL      ;VALUE*2
PUSH HL        ;SAVE VALUE*2
ADD HL,HL      ;VALUE*4
ADD HL,HL      ;VALUE*8
POP DE         ;VALUE*2
ADD HL,DE      ;VALUE*10
```

Every time we wanted to perform these six instructions, we'd have to key them in to the edit. Of course, one alternative would be to include them as a subroutine, but another way would be as a macro.

Figure 3-2 shows the six instructions **defined** as a macro by a **macro definition**, and later **invoked** as a macro by a **macro call**. The macro definition involved the pseudo-op **MACRO**, which defined the label **MUL10** as the macro name, and **ENDM** as the pseudo-op which ended the definition. After the definition, any use of the macro name in the op-code column would automatically generate a **macro expansion** for the six instructions. You can use the macro as many times as needed.

```

00100      ; SAMPLE MACRO USE
00110      ; FIRST, DEFINE THE MACRO
00120      MACRO
00130          AOO      HL,HL
00140          PUSH    HL
00150          AOO      HL,HL
00160          AOO      HL,HL
00170          POP      OE
00180          ADD      HL,DE
00190          ENOM
00200      ; NOW THE MACRO CAN BE INVOKED AS REQUIRED
00210      START:  LO    A,23H      ;BLAH, BLAH
00220          MUL10      ; INVOKE
          AOO      HL,HL
          PUSH    HL
          AOO      HL,HL
          ADD      HL,HL
          POP      OE
          ADD      HL,DE
          ENO
00230
0000'      3E 23
0002'      29
0003'      E5
0004'      29
0005'      29
0006'      D1
0007'      19

```

Figure 3-2. Simple Macro Use

Such macro use does have certain advantages as well as disadvantages. By defining the six instructions as a macro, we've made it much easier to generate the code simply by a macro call in the op-code. The code generated is also somewhat faster than the equivalent subroutine would be, since the overhead of the CALL and RET is gone. On the other hand, we've used up quite a bit more memory than the corresponding subroutine.

If this were the extent of macro capability, you might be tempted to forget the whole thing. However, you can use macros with **arguments** to generate **tailored** code to fit the generalized case.

An example of such a macro is shown in Figure 3-3. This macro will take a given character and fill a screen line with it. The macro definition consists of the macro name and MACRO pseudo-op as before, but it also has two **dummy arguments**, CHAR and LINENO, representing the character and the line number, 0-15, to be used.

```

00100      ; THIS IS A SAMPLE MACRO WITH ARGUMENTS
00110      ; FIRST DEFINE THE MACRO WITH OUMMY ARGUMENTS
00120      FILL    MACRO  CHAR,LINENO
00130          LOCAL  LOOP
00140          LO      A,CHAR
00150          LO      HL,3C00H+LINENO*64
00160          LO      B,64
00170      LOOP:  LD      (HL),A
00180          INC      HL
00190          DJNZ     LOOP
00200          ENOM
00210      ; NEXT, INVOKE THE MACRO WITH REAL ARGUMENTS
00220      FILL      ' ',0

```

THIS PREVENTS DUPLICATE LABELS WHEN MACRO IS EXPANDED

0000'	3E 20		LD	A, ' '	} EXPANSION 1
0002'	21 3C00		LO	HL, 3C00H+0*64	
0005'	06 40		LO	B, 64	
0007'	77	..0000:	LD	(HL), A	
0008'	23		INC	HL	} EXPANSION 2
0009'	10 FC		DJNZ	..0000	
		00230	FILL	'*', 1	
000B'	3E 2A		LD	A, 'B'	
0000'	21 3C40		LO	HL, 3C00H+1*64	} EXPANSION 2
0010'	06 40		LD	B, 64	
0012'	77	..0001:	LO	(HL), A	
0013'	23		INC	HL	
0014'	10 FC		DJNZ	..0001	} EXPANSION 3
		00240	FILL	88H, 2	
0016'	3E 88		LO	A, 88H	
0018'	21 3C80		LO	HL, 3C00H+2*64	
001B'	06 40		LD	B, 64	} EXPANSION 3
001D'	77	..0002:	LO	(HL), A	
001E'	23		INC	HL	
001F'	10 FC		OJNZ	..0002	
		00250	END		

Figure 3-3. Macro Use With Arguments

The arguments are called **dummies** because they only serve to denote when you are to use "real" arguments when the macro is called. When the macro is invoked, as shown in the listing, "real" arguments replace the dummy arguments, and specialized code is generated with the real arguments put in the proper places in the code. Any number of dummy arguments can be used in a macro, subject only to line length.

It is not hard to see how to create macros to define an entire applications language. The resulting code for the application could consist primarily of macro calls, allowing quick development time but at the expense of memory.

—Hints and Kinks 3-4— More Complicated Macros

The macros used as examples in the text are quite straightforward. The Disk Assembler, though, has a number of macro operators, conditional assembly pseudo-ops, and rules for macro definition and use that may be used to construct very powerful macros. Such things as a NUL operator, to test for a null argument, EXITM, to terminate a macro before a complete expansion, and SET, to permit redefinition of a variable name are but a few of the functions that can be used to create truly horrendous code!

Pseudo-Ops For Program Sections and Conditional Assembly

The Disk Assembler adds a number of new pseudo-ops other than those for EXT(ernals), ENTRYs, and macros. Some of these are related to **program sections** and some to **conditional assembly**.

Program Sections

The pseudo-ops ASEG, DSEG, and CSEG are related to **program sections**. The Disk Assembler is initialized in the "relocatable" mode; any code produced in this mode is relocatable and can be modified by the loader at load time to run wherever it is being loaded.

If the ASEG pseudo-op is used, however, all code following will be in **absolute mode**. It will be assembled to run at one specific area in memory. The ASEG is normally followed by an ORG (Origin) to define the area of memory at which the code is to be assembled.

```
ASEG                                ;SET ABSOLUTE  
ORG      8000H                      ;START OF ABSOLUTE AREA
```

The CSEG pseudo-op defines a **code relative** section of the program. This is the relocatable code section. CSEG is not required unless an ASEG has been used, in which case you should use a CSEG to force the Assembler back into its default mode. You can intermix ASEGs and CSEGs as needed.

The third pseudo-op of this type is the DSEG, or **data relative** segment definition. Here again, DSEG should be followed by an ORG to define where the data is to go. The DSEG could be used, for example, to include all variables and working storage in each program module in a separate data-segment area of memory for convenience in debugging.

Essentially, the Disk Assembler maintains three separate location counters — one for the absolute, one for the code

relative, and one for the data relative sections of memory. An example is shown in Figure 3-4, where the three types of program sections are used with relative impunity.

0000'	00100		
	00110		;THIS IS ABSOLUTE SECTION. USED TO SPECIFY ABSOLUTE CODE.
	00120		ASEG
8000	C3 8000		ORG 8000H
	00130		LOOP1: JP LOOP1
	00140		;THIS IS DATA SECTION. USED TO SPECIFY A DATA SECTION FOR
8003			;VARIABLES AND DATA.
	00150		ASEG
	00160		ORG 0C000H
C000"	01		DATA1: OFEB 1
	00180		;THIS IS CODE RELATIVE. USED TO SPECIFY "NORMAL" RELOCATABLE
	00190		;CODE.
	00200		CSEG
C001"			LOOP2: JP LOOP2
0000'	C3 0000'		;ANOTHER OSEG. NOTE ADDRESS IS ONE MORE THAN LAST OSEG.
0003'			ASEG
C001"	02		DATA2: OFEB 2
	00250		;ANOTHER ASEG. NOTE ADDRESS IS AT NEXT ABSOLUTE ADDRESS.
	00260		ASEG
C002"			LOOP3: JP LOOP3
8003	C3 8003		;ANOTHER CSEG. NOTE ADDRESS IS AT NEXT CODE RELATIVE
			;ADDRESS.
8006			CSEG
0003'	C3 0003'		LOOP4: JP LOOP4
			END
	00330		

Figure 3-4. ASEGs, CSEGs, and DSEGs

Conditional Assembly

Another set of pseudo-ops are concerned with **conditional assembly**. Conditional assembly refers to a segment of code that is either assembled or not assembled according to some defined condition. Conditional assembly might be used, for example, by a company that has many versions of a software program. One segment of code might be necessary if a user has a disk system, while another might be necessary if the user has cassette. The section for a disk system might appear as follows:

```
    ;ASSEMBLE THIS CODE FOR DISK
        IF      DISK      ;DISK FLAG
DISK  LD      A,5      ;LOAD FLAG
        .
        .
        .
    ENDIF
```

If variable DISK is not zero, this code will be assembled. If DISK=0, the code will not be assembled. The dots indicate additional code in the segment. There are a number of conditional assembly pseudo-ops including the REPT ("repeat n times") and IRP pseudo-ops. For information on the rest, consult the *Disk Assembler Programming Manual*.

There are a number of pseudo-ops we haven't talked about here, such as SET, which allows an assembly time variable to be redefined, REPT, which allows a block of statements to be repeated n times; and LOCAL, which defines a local label within a macro.

In my opinion, most assembly-language programmers probably won't exercise the full capabilities of the Disk Assembler by using such 'elegant' operations. The last time I used an elaborate scheme to have the assembler generate a large complicated table of data, I was awakened at 2:00 a.m. by a call from the computer operator. The operator informed me that my (200-line) assembly had been assembling for 4 hours and asked if it was 'normal'!

Pseudo-Ops for Listing Format

There are a number of pseudo-ops in the Disk Assembler that make listings "prettier." TITLE specifies a title to be used on each page of the listing. SUBTTL allows the user to specify a subtitle. PAGE causes a "page eject" to start the listing on a new page; this is handy for isolating logically different sections of code. .COMMENT may be used in place of comment lines starting with a semicolon. The first character of the following text is used as a **delimiter** and the following text is used as a comment block until the same character is encountered.

Several other pseudo-ops control listing output, macro listing output, and **cross-reference** listing output. See the Disk Assembler manual for details.

Using the Disk Assembler

After you have used the Editor to create assembly-language source modules, the Assembler reads in a source module from disk, assembles the code, and optionally produces a relocatable object module and **listing file**.

The format for the assembly command is —

```
*NAME1 ,NAME2=NAME3  
* ,NAME2=NAME3  
*NAME1 ,=NAME3 or  
*=NAME3
```

In case this format's confusing (I found it so), we'll interpret for you. The first command assembles source file, NAME3, from disk and produces a relocatable object file, NAME1, and a listing file, NAME2, on disk. The listing file may be listed by a PRINT command.

The second command does *not produce a relocatable object-file*. The third command does *not produce a listing file*. The fourth command produces *neither* a relocatable object file *nor* a listing file (handy for assembly error checks).

The names may actually be the same for all three. If a source file is on disk called NAME/MAC, the command

```
*NAME ,NAME=NAME
```

produces a relocatable object file called NAME/REL and a listing file called NAME/LST. The file extensions are automatically generated by the assembler. Of course, you can also use passwords (ugh!) or disk drive numbers.

CREF

We haven't mentioned the cross reference facility of the Editor/Assembler at all. It's not that it isn't useful, but it does create another step in the assembly process. You must set the CREF switch during assembly by an assembler command such as *TEST,TEST/C=TEST. The resulting listing file has 'hooks' in it that enable processing by CREF. The result is an alphabetized listing of the variable names along with line numbers of lines in which each is referenced or defined. Use at your option.

Loading the Object Modules to Produce a Command File

Let's assume we've gone through the assembly process for the three modules shown in Figure 3-5. They make up a horrendously complicated program to clear the screen, print a title, read a key of 0-7, and output the key to the center of the screen. Three modules were used here so we could illustrate the linkages between the modules at load time.

00040		TITLE	READ
00050		ENTRY	
00100		; READ ROUTINE. READS KEYS 0-7 AND OISPLAYS.	
00110		READ:	LD A,(3810H)
00115			;GET 0-7
00120		OR	A
00130		JR	Z, READ
00140		LD	B,-1
00150		READ10:	INC B
00160			;BUMP COUNTER
00170		RRCA	
00180		JR	NC, READ10
00190		LD	A,B
00200		A00	A,30H
00210		LD	(3820H),A
00220		RET	
00230		END	

00100		TITLE	PRINT
00110		ENTRY	
00120		; PRINT ROUTINE. PRINTS TITLE	
00130		PRINT:	LD HL,3C00H
00140		LD	DE,MSG1
00150		LD	A,(0E)
00160		OR	A
00170		RET	Z
00180		LD	(HL),A
00190		INC	HL
00200		INC	DE
00210		JR	NC,PRINT
00220		MSG1:	DB 'TITLE',0
00230		END	

00100		EXT	PRINT, HEAD
00110		; MAIN DRIVER ROUTINE. CLEARS SCREEN, PRINTS MESSAGE	
00120		; AND HEADS AND DISPLAYS KEY BY CALLING PRINT, READ.	
00130		START:	LD HL,3C00H
00140		LD	DE,4000H
00150		LD	A,' '
00160		LD	(HL),A
00170		INC	HL
00180		OR	A
00190		SBC	HL,DE
00200		A00	HL,0E
00210		JR	NZ,LOOP1
00220		CALL	PRINT
00230		LOOP2:	CALL READ
00240		JR	LOOP2
00250		END	

0000'	3A 3810
0003'	B7
0004'	28 FA
0006'	06 FF
0008'	04
0009'	0F
000A'	30 FC
000C'	78
000D'	C6 30
000F'	32 3E20
0012'	C9

0000'	21 3C00
0003'	11 000E'
0006'	1A
0007'	B7
0008'	C8
0009'	77
000A'	23
000B'	13
000C'	18 F8
000E'	54 49 54 4C
0012'	45 00

0000'	21 3C00
0003'	11 4000
0006'	3E 20
0008'	77
0009'	23
000A'	B7
000B'	E0 52
000D'	19
000F'	20 F8
0010'	C0 0000*
0013'	CD 0000*
0016'	18 FB

Figure 3-5.
Loading Object
Modules

The modules are out on disk as MAIN, PRINT, and READ and emulate the usual arrangement of modules produced while using the Disk Assembler. The main portion of code calls the other modules as subroutines. There could also be ENTRY points in MAIN or EXTERNALS in PRINT and READ if this were more complicated code.

The Loader is entered by typing L80 after the TRSDOS prompt, TRSDOS READY. The chief commands to consider in the Loader are the -P command, the -N command, and the -E command. The -P command has the form -P:XXXX. It sets the loader location counter to location XXXX. The -N command has the form NAME-N. It establishes a name for the command file. The -E command ends the edit, outputs the linked relocatable modules to the disk as command (/CMD) file NAME, and ends the edit.

The load sequence of commands is shown in Figure 3-6. The load location counter is first set to 8000H by a -P command. The three modules are then loaded by typing in each name. Finally, the linked modules are output to disk as TEST/CMD, and the load is ended. TEST can now be loaded and executed from TRSDOS by simply typing TEST.

*-P:8000				SET LOAD ADDRESS
*MAIN				LOAD MAIN MODULE
DATA	8000	8018		SIZE OF CURRENT DATA
PRINT*	8011	READ*	8014	UNRESOLVED EXTERNS
*PRINT				LOAD PRINT MODULE
DATA	8000	802C		SIZE OF CURRENT DATA
READ*	8014			UNRESOLVED EXTERNS
*READ				LOAD READ MODULE
DATA	8000	803F		SIZE OF CURRENT DATA
*TEST-N, -E				FILE NAME "TEST", END
(0000	803F)			
DOS READY				

Figure 3-6. Typical Load Commands

Figure 3-6 also shows the loader symbol table displayed after each load. This lists all unresolved symbols in addition to the memory area loaded. At the end of the load, all externals should have been resolved, or a load error will result. If any external is unsatisfied, it means that a module referenced an external name by an EXtern, and there was no corresponding ENTRY point. This may mean that you did not load the module containing the name, or that you didn't declare the name in an ENTRY. Either condition is an error.

The procedure above is the loading process in microcosm for any size assembly-language program. The Tic-Tac-Toe program in Chapter 14 uses 25 separate modules with a large number of ENTRY points, yet the procedure is virtually identical.

The "core image" of the three modules above is shown in Figure 3-7. All addresses have been satisfied during the load process, and the program appears as one contiguous program block.

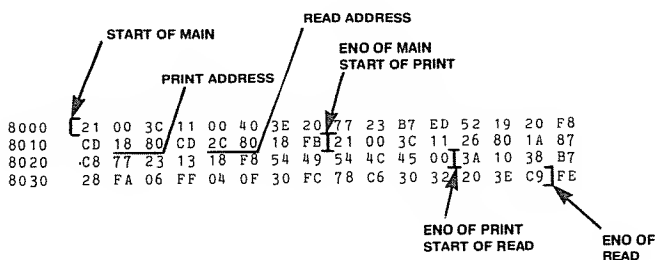


Figure 3-7. Core Image Module

Chapter Four

Loading, Executing, and Debugging Assembly-Language Programs

Up to this point we've talked about general considerations in producing programs using EDTASM and the Disk Assembler — how to edit, assemble, and list source files. We'll move on to discuss the next step, loading and executing the resulting object code. You can run assembly-language programs as "stand-alone" programs or interface them to BASIC programs. This chapter describes both methods. We also cover "debugger" programs and some tricks in debugging techniques. In short, we're going to talk about the subject, "Now that I have the assembled program, what do I do with it?"

EDTASM System Tapes

The result of an edit and assembly using EDTASM is a SYSTEM file on cassette. A SYSTEM file resembles the machine-language code seen on the EDTASM listing. It contains other data, however, to enable the Level II BASIC interpreter to load the machine-language code into the proper place in memory and verify the data as correct. The format for SYSTEM tapes is shown in Figure 4-1.

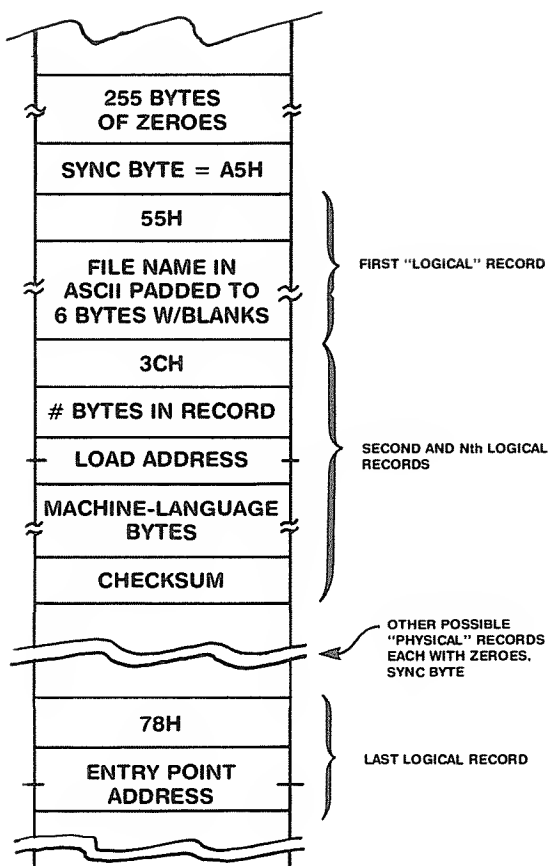


Figure 4-1. SYSTEM Tape Format

A string of 255 zeroes are at the beginning of the cassette file, followed by a **sync byte** of A5H. The sync byte synchronizes the cassette tape driver of the BASIC interpreter — informing it that the next byte will be valid data. The zeroes and sync byte precede each **physical record** on the SYSTEM tape. A SYSTEM tape may have one or more physical records, depending upon the size of the object file. Each physical record is divided into a number of **logical records**, which are essentially **load item** blocks defining file name, data, or entry point. After the sync byte of the first logical record, a file name code of 55H occurs. This informs the interpreter that a file name will appear in the

next 6 bytes. The file name is 6 bytes long, padded out with blanks. This is the first logical record of the SYSTEM tape.

The next logical record is headed by a data code of 3CH. This informs the interpreter that machine language data follows in the record. The next byte in the data record is a count of the number of data bytes in the record. If this byte is a zero, the number of data bytes is 256, otherwise the count is the actual number. The next two bytes are the load address for the data to follow. They're set in standard address format, least significant byte followed by most significant byte. Next are the machine-language bytes. The last byte of each data record is a **checksum** byte. A checksum is nothing more than a "check byte" formed by adding all of the individual data bytes. It is used for comparison to another add of the data bytes on load to verify that the data has been loaded properly.

The last logical record on the SYSTEM tape is an entry point code of 78H and two bytes that represent the entry point for program execution.

Hints and Kinks 4-1

An Actual Cassette Object File

The data below represents an actual cassette SYSTEM file produced by assembling the program below onto cassette.

```

C000      08760      ORG      0C000H
          08770 ;
          08780 ;*****FILL CHARACTER SUBROUTINE*****
          08790 ; FILL DESIGNATED AREA WITH GIVEN CHARACTER
          08800 ; ENTRY: (A)=CHARACTER
          08810 ; (DE)=AREA
          08820 ; (BC)=NUMBER OF BYTES, 1-65525; 0 IS 65536
          08830 ; ALL REGISTERS SAVED EXCEPT EC,DE
          08840 ;
C000 12      08850 FILLCH LD      (DE),A      ;FILL CHARACTER
C001 13      08860 INC      DE                ;BUMP POINTER
C002 0B      08870 DEC      BC                ;DECREMENT COUNT
C003 F5      08880 PUSH     AF                ;SAVE FILL CHAR
C004 78      08890 LD      A,B                ;TEST FOR ZERO
C005 B1      08900 OR       C
C006 2B03    08910 JR      Z,FILO10          ;GO IF DONE
C008 F1      08920 POP      AF                ;RESTORE FILL CHAR
C009 1BF5    08930 JR      FILLCH            ;CONTINUE
C00E F1      08940 FILO10 POP      AF         ;RESTORE A
C00C C9      08950 RET                ;RETURN
C000      08960 END      FILLCH
C0000 TOTAL ERRORS

```

Diagram illustrating the structure of the cassette object file:

SYNC BYTE | FILE NAME CODE | DATA CODE | CHECKSUM | ENTRY POINT CODE

NAME="FILL 55"

DATA BYTES FOLLOWING LOAD ADDRESS = C000

12 13 0B F5 78 B1 2B 03 F1 18 F5 C9 F1 78 00 C0 (END)

MACHINE-LANGUAGE CODE

ENTRY POINT = C000

The SYSTEM tape format is identical to that produced by T-BUG, the **debugger** program for cassette-based systems. T-BUG has the capability of producing such a file by the P (punch) command. Conveniently enough, T-BUG can also load in the cassette file by the L command. This means that the object tape produced by EDTASM can either be read by using the BASIC SYSTEM command or by T-BUG!

System Considerations for EDTASM Object Files

EDTASM object files represent one huge program. The start of the program is defined by the ORG (Origin) pseudo-op at the beginning of the program. The size of the program can be determined by the location column on the listing. You can put buffer areas and **working storage** for variables and tables anywhere in the program that you want. There may be a large "open-ended" buffer or buffers in the program, generally at the end of the program to build "up" into higher and higher memory. You can see an example of this structure in MORG, the EDTASM program of Chapter 13.

What should the value be for Origin? This depends upon the environment in which you are going to use the assembly-language program. If your system does not have disk, and the assembly-language program does not interface to a BASIC program, then the program may be ORGed at 4980H. T-BUG occupies the area from about 4380H through 497FH, with an internal stack area building downwards from 497FH. When the program is debugged using the T-BUG stack, the program may use all of memory for storage from the end of the program up to **top of memory**.

You might be curious about establishing your own stack area. You can do it. In this case, set aside approximately 100 bytes at some convenient point either **in** your program, at top of memory, or in some area of memory that you know won't be used. The stack should be initialized immediately with a

```
START LD SP,PEND+100
```

or similar instruction (here the stack has been set to the end of the program plus 100 bytes). Remember, the location loaded into the stack pointer represents the first location to be used by the stack **plus one**.

Hints and Kinks 4-2

Size of Stack

Many programmers ask "How big should the stack area be?" There's no definitive answer. Each time a CALL is made, the two bytes of the return address are pushed onto the stack. Each time a PUSH instruction is executed to temporarily store data, two bytes of a register pair, IX, or IY are pushed onto the stack. A third function that pushes data onto the stack is interrupt action. This can occur in a disk system with the real-time-clock enabled. The "worst case" number of bytes used, therefore is:

$2 * (\text{number of calls active at any time} + \text{maximum number of pushes active at any time} + \text{possible interrupt})$

If 100 bytes are allocated, this allows for 50 CALL/PUSH/interrupt levels, which are probably more than adequate.

The configuration for assembly-language programs in this type of environment is shown in Figure 4-2.

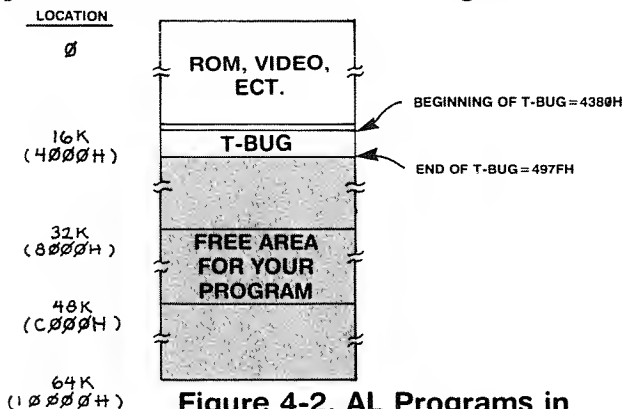


Figure 4-2. AL Programs in Minimum Configuration Systems

If the program is to be used together with a BASIC program in a non-disk system, then there are two considerations. First of all, the ORG can't overlap the BASIC program area. The BASIC program builds up in memory from about 4200H on. The program statements aren't the only things occupying memory, however. Simple variables, arrays, strings, and a BASIC interpreter stack are also stored, as shown in Figure 4-3.

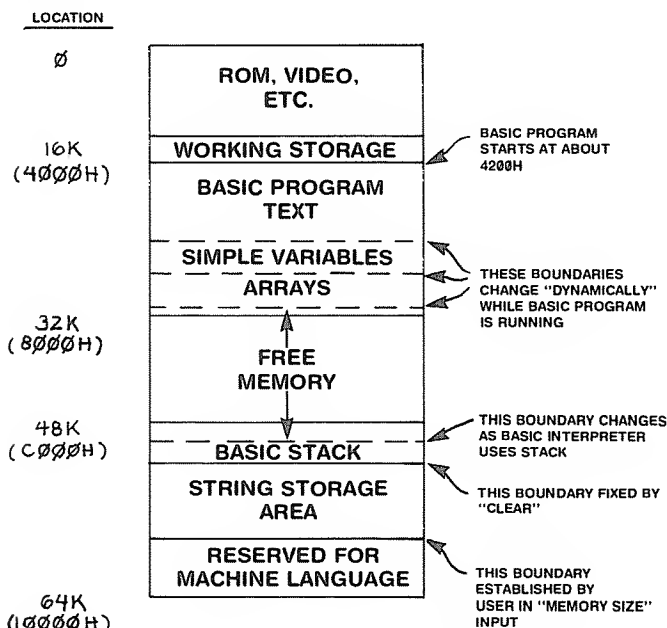


Figure 4-3. BASIC Memory Allocation

Compute the ORG for the assembly by taking the size of the assembled program plus any buffer areas or tables outside of the program and subtracting it from your system's top of memory plus one (8000H for 16K, C000H for 32K, or 10000H for 48K). You may use the BASIC stack with no problem; if you establish your own stack area, add this into the total size to be subtracted.

Assemble the program with this Origin, and protect this memory area by entering the Origin value for MEMORY SIZE when BASIC is first entered. The stack area

for BASIC will now build downwards from the Origin minus one location.

A special case arises when the assembly-language program uses machine code **embedded** in the BASIC program. In this case, a short assembly-language program becomes part of the BASIC program itself and can be loaded easily. We'll talk about these techniques in the next chapter.

Debugging With T-BUG

After you edit, assemble, and thoroughly **desk-check** your assembly-language program, you should load it along with Radio Shack's cassette based debugging program, T-BUG. The procedure is simple:

1. Enter Level II BASIC.
2. Enter the area of memory to be protected (MEMORY SIZE), computed by the procedure above.
3. Type SYSTEM for the ">" prompt. This causes the BASIC interpreter to enter the **monitor** mode.
4. Rewind your object tape and prepare for a load.
5. Enter NAME after the *? prompt. NAME is the name you've given your object file for assembly. If you haven't named your file, the cassette file will be called NONAME.
6. Your object tape should now load, as indicated by the blinking asterisk.
7. After a successful load, ready the T-BUG cassette.
8. Enter TBUG after the *? prompt.
9. The T-Bug file should now load, followed by the *? prompt.
10. Enter / after the prompt.
11. T-BUG should now be entered, as evidenced by a # prompt.

With both T-BUG and your object program in memory, you're ready to do some serious debugging. (This procedure

can't be used when a BASIC program is also resident, as the BASIC program will "overlay" T-BUG.) First of all, you should be familiar with T-BUG commands. These are described in the T-BUG manual, and we won't study them here, but we'll tell you how to make efficient use of them.

—Hints and Kinks 4-3—
Recap of T-BUG Commands

#M aaaa	Display location aaaa
ENTER (after M)	Display next location
X (after M,J,B,P)	Exit operation
#R	Display registers
#P aaaa bbbb cccc	Write cassette from aaaa
NAME	through bbbb with starting
	address cccc and file name
	NAME
#L	Load a T-BUG or SYSTEM tape
#B aaaa	Set breakpoint
#F	Restore instruction after
	breakpoint
#G	Continue from breakpoint
#J aaaa	Jump to location aaaa

The first rule of debugging is to be sure you've thoroughly **desk checked** the program. This means you've taken a listing of the program and gone over it minutely, instruction by instruction, to see that it works as planned. This may mean some reference to flow charts and design specs with larger programs.

After desk checking as much as possible, the procedure is basically, "Get it working (even though it probably will run badly) then go back and clean up the flaws." This procedure works well as long as the program design is substantially correct, and there isn't an overabundance of errors.

To get the program working, use a type of "binary search" for flaws. First of all, go ahead and give it a try. Chances are that the program will blow up, but it takes an extremely strong-willed programmer not to try that first

execution in hopes that everything will work right off the bat. (It never does!) Reload the program if necessary.

Having gotten that out of your system, set a breakpoint about half way through by using the T-BUG B(reakpoint) command. If, for example, the program is about 100 locations long, set a breakpoint at a location 50 bytes from the start of the program. Now use the J(ump) command to start the program. One of two things will happen. The program may "bomb" again, necessitating a reload, or the breakpoint will be reached.

If the breakpoint is reached, F(ix) the breakpoint and start looking at variables and "tracks" of the program up to that point. See that variables and actions appear to be correct. If you find strange results, write them down and start correcting them one by one by setting new breakpoints at an earlier condition. Don't be too picky about testing every possible condition.

— Hints and Kinks 4-4 — How Does T-BUG Breakpoint?

When you put a breakpoint at a specific location, T-BUG puts a CD 80 43 at that location and the next two bytes. This is a 'CALL 4380H' that calls the T-BUG breakpoint routine. F(ixing) the breakpoint restores the original three bytes.

Be careful in breakpointing that the three bytes temporarily destroyed in breakpointing are not used as variables or instructions by the code to be checked.

If results seem to be correct on cursory looks up to this point, set a breakpoint about half-way into the remaining area, and repeat the procedure. **Zero-in** on the errors in the same fashion.

Patching

You have two alternatives in correcting errors as they're

found in the program. You can reassemble and reload, or you can **patch**. If you have never used a patching technique; stay tuned and we'll explain with an example.

Figure 4-4 shows an EDTASM program from MORG in Chapter 13 with two errors. The subroutine is missing an OR C, and the JR Z,FIL010 is erroneously JR NZ,FIL010. (Obviously I had a thorough desk check here!)

```

8583      08770      ORG      8583H
08780      ;*****FILL CHARACTER SUBROUTINE*****
08790      * FILL DESIGNATED AREA WITH GIVEN CHARACTER *
08800      * ENTRY: (A)=CHARACTER *
08810      * (DE)=AREA *
08820      * (BC)=NUMBER OF BYTES, 1-65525; 0 IS 65536 *
08830      * ALL REGISTERS SAVED EXCEPT BC,DE *
08840      ;
5583 12      08850  FILLCH LD      (DE),A      ;FILL CHARACTER
5584 13      08860      INC      DE           ;BUMP POINTER
5585 0B      08870      DEC      BC           ;DECREMENT COUNT
5586 F5      08880      PUSH     AF           ;SAVE FILL CHAR
5587 78      08890      LD      A,B          ;TEST FOR ZERO
08900      ; OR      C                      ;***MISSING***
5588 2003    08910      JR      NZ,FIL010      ;GO IF DONE***S/B Z***
558A F1      08920      POP      AF           ;RESTORE FILL CHAR
558E 18F6    08930      JR      FILLCH        ;CONTINUE
558D F1      08940  FIL010 POP      AF           ;RESTORE A
558E C9      08950      RET                  ;RETURN
08960      ;
0000      08970      END
00000 TOTAL ERRORS

```

Figure 4-4. Flawed Program

In testing the program, we loaded the A register with 23H, the DE register pair with 8000H, and the BC register pair with 64H to fill locations 8000H-8063H with 23H. We breakpointed on the RET by B58E. When the breakpoint was reached, it turned out that only the first byte at location 8000H had been filled.

In checking through the code, we found that the JR NZ,FIL010 should have been a Z. Rather than reassembling, we decided to patch the code. The JR NZ,FIL010 assembles as 2003H. By looking in Appendix II (or the Assembler manual), we found that a JR Z would be 28XXH. This is simple to fix using T-BUG — we simply perform a M 5588 20 28 to change the location to a JR NZ,FIL010 and record the patch in a list of patches.

Having patched, we tried again. This time, with an identical procedure, we found 23H stored from 8000H through

8583H! It seems 23H was stored until the program was destroyed! After further head scratching, we found that we had left out an `OR C` instruction!

This time we found we couldn't easily patch because we couldn't fit in another one-byte instruction between the last **bit** of location 8587H and the first **bit** of location 8588H! How do we patch in this case?

In cases like this, we can jump out to a **patch area**, "gin up" some code to make the change, and jump back into the routine. A patch area is any unused area that can be used to store temporary fixes. In this case, since we weren't using locations C000H and up, we designated that as a patch area.

To get to the patch area, we had to put the three bytes of a `JP` somewhere in the code. We put them into the locations that would have normally held `DEC BC` as shown in Figure 4-5. We deleted the three instructions of `DEC BC`, `PUSH AF`, and `LD A,B` and substituted a `JP C000H` instead.

<u>Location</u>	<u>Old</u>	<u>Patched</u>
8583	12	12
8584	13	13
8585	0B DEC BC	C3 JP C000H
8586	F5 PUSH AF	00
8587	78 LD A,B	C0
8588	28 03	28 03
858A	F1	F1
858B	18 F6	18 F6
858D	F1	F1
858E	C9	C9

PREVIOUSLY
PATCHED
JR2, F1010

Figure 4-5. Patching to Flawed Program

How did we know what code to put in for the JP? By consulting the Appendix, by referring to the Assembler manual, or by **looking through the listing until a similar type instruction was found!**

At the C000H patch area, we put in machine code representing the three deleted instructions, machine code for OR C, and a JP back to location 8588H, as shown in Figure 4-6. You can patch virtually any program this way.

Hints and Kinks 4-5

Hand Assembling

There are still some programmers that I know of that persist in "hand assembling." These are the same types of people that collect balls of string eight feet in diameter, cultivate their own wheat in their city plot, and try to interface Baudot teletypes to their TRS-80s!

What we are doing in patching is a small exercise in "hand assembling." Instead of letting the assembler look up opcodes and resolve addresses, we are doing it ourselves. Hand assembling is not too much of a chore in this case, since we're working with only a few instructions. We can simplify it by looking at our listings for identical or similar instructions and use their formats to make the patches. However, when there are many instructions to be assembled, it's probably best to let the assembler do the job - even to the extent of running a short assembly to get the patches! I've spent too many nights at some customer's site hand assembling patches into a computerized system to advise you differently. Help stamp out hand assembling!

<u>LOCATION</u>	<u>CONTENTS</u>	<u>CODE</u>	<u>NOTES</u>
C000	0B	DEC BC	} RESTORE ORIGINAL INSTRUCTIONS
1	F5	PUSH AF	
2	78	LD A,B	
3	B1	OR C	} NEW INSTRUCTION
4	C3	JP 8588H	
5	88		} JUMP BACK
6	85		

Figure 4-6. Patch Area Contents

With the patch in place, we tried the subroutine again, and it worked fine for the one case.

This patching process requires some effort, and you may not want to use it. However, you can become very proficient at it and use it to advantage to debug large programs that take a great deal of time to reassemble because of printing time. Hard thinking vs. time: it's a tradeoff you'll have to assess.

A patched program can be written out to cassette at any time. The advantage is that the patches don't have to be reentered when the program **blows up** the next time. Also, as long as you're writing out to cassette, why not include the T-BUG area along with the program area? That way the entire program area, T-BUG and all, can be read in by a SYSTEM command to simplify reloading. The entry point for T-BUG is 17312 decimal.

Disk Assembler Files

The final output of a set of edits and assemblies and a load operation with the Disk Editor/Assembler package is a **command** file (/CMD) on disk. You can load and execute this command file by simply typing the name of the file after the DOS READY prompt. In most cases, however, you must do some debugging before the program is ready to run in this fashion.

The memory area specified to the Loader is, like EDTASM programs, dependent upon the environment in which you are going to run the program. Since the program will

normally be loaded by TRSDOS, it must not overlap the TRSDOS area, which usually ends at 6FFFFH. This will enable DUMP commands to be used to save **patched** versions of the program.

If the program is to run "stand-alone," without interface to Disk BASIC, then the area from 7000H through top of memory is available for program use. This won't conflict with use of the Disk DEBUG package, as it loads into an **overlay** area below 7000H. If DEBUG were used for debugging, its stack area would be internal to the DEBUG program and wouldn't conflict with program use of the 7000H and up area. Of course, a separate stack area could be maintained by the program if you want. This configuration is shown in Figure 4-7.

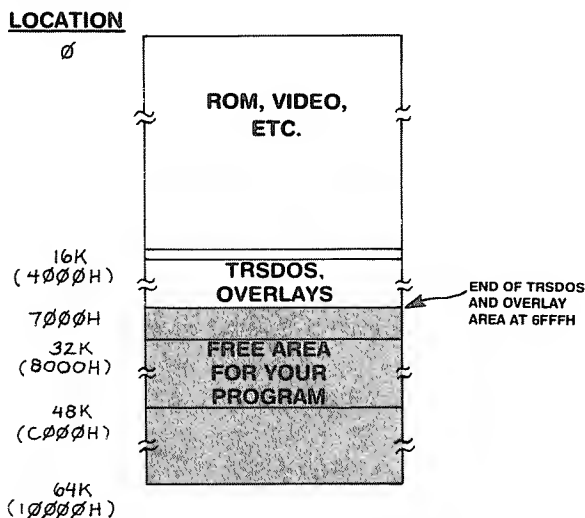
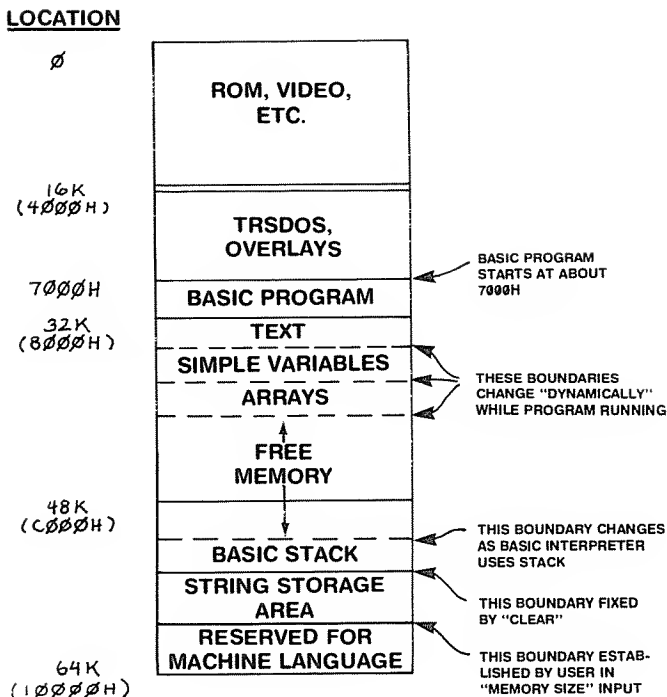


Figure 4-7. AL Programs in Disk Systems Without BASIC Interface

If you're going to use the program together with Disk BASIC, then the area from 7000H on is used for storage of the BASIC program, simple variables, arrays, strings, and the stack as shown in Figure 4-8. This allocation scheme is identical to Level II BASIC, except that the memory allocation area starts higher in memory.



**Figure 4-8. AL Programs in
Disk Systems With BASIC
Interface**

The plan here is for you to specify the load address high enough in memory for both BASIC and the assembly-language program to coexist peacefully. Compute the loading address by making a test load and observing the size of the final configuration. Add to this size any external buffer or table areas and approximately 100 bytes for an external stack, if one is to be used. Subtract this total from the top of memory value plus one. The result is the load address you'll specify to create the command module for the program. This process is shown in Figure 4-9. In many cases, of course, you'll have more than enough memory to be fairly "sloppy" in specifying the load address.

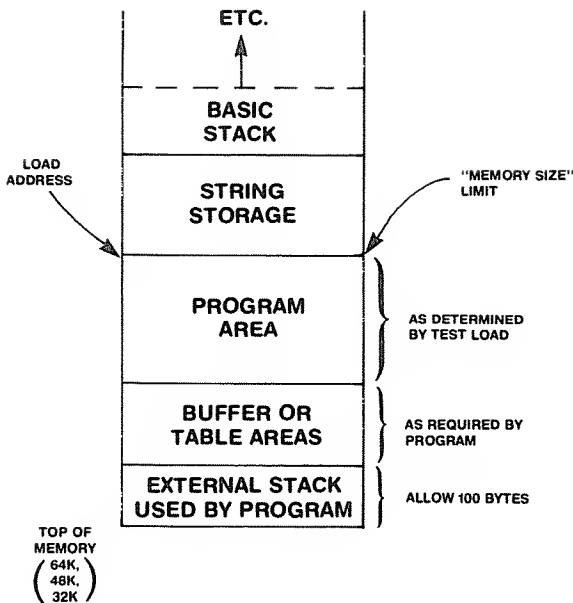


Figure 4-9. Computing the Load Address

Debugging with DEBUG

Once you've created a command module, you're ready to begin debugging. Actually, you should have already done the bulk of the debug! Prior to using DEBUG, you should have done a thorough and detailed check of your assembly-language code. This **desk check** will usually reveal errors in **logic** and implementation that will be much easier to correct by reassembly than by **patching** during debugging.

Desk Checking

A typical desk check would involve reviewing the specs and flow charts, together with the program listing(s) to see that no logical errors have been made.

Next, you might start with the simplest subroutines and 'walk through' them, instruction by instruction, using sample data. Write the contents of each register down on paper as you follow the instructions. Work your way up through the program levels until you've gone over all the code once. Reassemble if necessary.

Next, go through the same process again. Look for errors! They're there. (It might help to pretend that it's someone else's code.) This process should be repeated until you've gone through all code once without finding new errors. Yes, this is a lot of work. But it's much easier to correct errors now than during program execution.

When you're satisfied, start the actual 'on-line' debugging. You'll probably say 'What a stupid error - I should have seen that!' (It always happens to me)

To use DEBUG, first load DEBUG by typing in DEBUG after the TRSDOS DOS READY prompt. Then type in the name of the load module you created during the loading process to load it into memory in the area you specified. Now hit BREAK, and you will be in DEBUG.

The Disk DEBUG is quite a bit more powerful than T-BUG. Probably one of its most useful features is the ability to set **multiple breakpoints**. T-BUG allowed only one breakpoint, making it fairly easy for the program to take a path to oblivion. In addition to breakpointing, DEBUG allows for modifying memory or registers, displaying areas of memory in ASCII or hexadecimal, and **single-stepping**

the program one instruction at a time. These functions are described in the TRSDOS and Disk BASIC manual, and we'll be giving you some more examples of their use here.

— Hints and Kinks 4-7 —
Recap of DEBUG Commands

A	Show display in ASCII
C	Single step instruction, CALLs in full
Daaaa<space>	Display from location aaaa
Gaaaa(,bbbb (,cccc))	Execute at location aaaa with optional breakpoints bbbb,cccc,...
H	Display in hexadecimal
I	Single step instruction
M(aaaa)<space>	Set modification address to aaaa and display data
Rrp dddd <space>	Load register pair rp with dddd
S	Set display to full screen
U	Set dynamic display update mode
X	Cancel command, set display to register mode
;	Increment memory display to next block
-	Decrement memory display to last block

The debugging process with DEBUG is very similar to the one described under T-BUG use. The approach is again to focus in on failing areas of the program by a type of "binary search."

Start execution of your program by entering a G(o) command to DEBUG, with one or more breakpoints specified. If the program gets to the breakpoint, go back and look for "tracks" of proper program operation — flags set properly, variables and buffers with correct values, and the like. If errors are found, concentrate on the failing area by setting a breakpoint prior to the error and observing results. If the program doesn't get to the

breakpoint, hit BREAK to get back to DEBUG and then try another execution with an earlier breakpoint.

The DUMP command can be used to advantage in **patching** programs. (If you skipped over the discussion of patching under T-BUG, you may want to read it to get familiar with the technique.) When your program has been patched, you can save it on disk by doing something similar to:

```
DUMP NAME (START=X'8000',END=X'9A45'),
```

This creates a new file on disk with name, NAME, and extension, /CIM (Core Image Module). You can now load this patched file eliminating a tedious patching operation each time the program "blows up."

— Hints and Kinks 4-8 —
How Do You Know Where A
Relocatable Program Is?

One of the problems in link loading is knowing where a location is. Subroutines can be found by looking at the load map listing of all globals (loader M command). Record this information after the load.

It's also easy to put in global symbols, not only for subroutines but also for other selected points in the program. These will then be displayed on the load map, and you'll avoid some hexadecimal arithmetic.

A second trick is to use the -P command to load each module at convenient locations such as 8000H, 8200H, 8400H, etc. This makes the hexadecimal arithmetic a little easier.

To use DUMP, G(o) to location 402DH. This reboots TRSDOS but does not affect any of the user memory area. Now enter the DUMP command with the proper memory limits specified. By the way, if you've patched **outside** your program area, be sure to include that area by altering the START or END address accordingly.

The DUMP command is best used with a "clean" program. When you have found an error and established the proper patch, reload the program; **do not execute** and make the patches and then DUMP the patched, unexecuted program. This will eliminate strange results caused by patching an altered program.

However, if you want, you can also use the DUMP to save the state of any program at any given time during execution. This enables you to load in the program and easily reconstruct the conditions that caused the blowup.

Interfacing Assembly-Language and BASIC Programs

Level II and Disk BASIC both have the capability of calling assembly-language programs while in the middle of BASIC program execution. This means that **time-critical** parts of the BASIC program can be coded in assembly language to speed up program execution, or that BASIC can be used for operations which are harder to code in assembly language, such as string input, report display, and "number-crunching."

Level II USR Calls

The Level II procedure for calling an assembly-language program is as follows:

1. POKE the address of the assembly-language routine to be called into locations 16526 and 16527. The two bytes represent the least significant and most significant bytes of the address in standard Z-80 address format.
2. Any time a call to the assembly-language routine is required, call the routine by a statement such as `A=USR(M)`.

The USR function uses two **arguments**, A and M. These can be "dummies." For example, a call to the assembly-language routine can be made by `A=USR(0)` with variable A being ignored. The USR function simply causes

the BASIC interpreter to pick up the address from locations 16526 and 16527 and execute a CALL to that address. The assembly-language routine must have a RET at the end to cause a return back to the interpreter at a point after the CALL. Figure 4-10 shows a typical Level II USR sequence for a short assembly-language routine located at location C000H.

```

100 REM CALL WRITE BOOK ROUTINE AT C000H
200 REM FIRST POKE LEAST SIGNIFICANT BYTE OF ADDRESS
300 POKE 16526,0
400 REM NEXT POKE MOST SIGNIFICANT BYTE OF ADDRESS
500 POKE 16527,192
600 REM NOW CALL ROUTINE BY USR FUNCTION
700 A=USR(N)
800 REM RETURN AFTER USR ROUTINE HERE

```

**Figure 4-10. Level II
USR Sequence**

You can make calls to more than one assembly-language routine easily by setting up 16526,7 with the proper address just prior to the USR call. You can use as many assembly-language routines as you need by preceding each call by a POKE of the address into 16526,7. Of course, if you use only one assembly-language routine, you only have to set up 16526,7 once at the beginning of the BASIC program.

The M argument allows the BASIC program to pass one 16-bit argument to the assembly-language routine. The calling sequence is the same except that M must be a variable (or expression) that is equated to -32768 to 32767. When the assembly-language routine is entered, it must pick up the argument by performing a CALL to ROM at 0A7FH. This CALL passes the 16-bit argument back in the HL register pair.

If you wanted to pass the argument back the other way — in this case, pass a 16-bit value in HL back to the BASIC program — you would make a JP 0A9AH at the end of the assembly-language program rather than a RET. Figure 4-11 illustrates the passing of a single argument back and forth; here, a variable passed to a shift routine. The shift routine shifts the value three bits left and passes back the result. The BASIC program then displays the result on the screen.

```

C000          00100      ORG      0C000H
              00110      ;SHIFT ROUTINE. SHIFTS CONTENTS OF HL LEFT 3 BITS AND
              00120      ;PASSES BACK THE RESULT
C000 C07FOA    00130      SHIFT    CALL    0A7FH      ;PICK UP ARGUMENT
C003 29        00140      AOO      HL,HL      ;SHIFT ONE BIT POSITION
C004 29        00150      ADO      HL,HL      ;SHIFT TWO BIT POSITIONS
C005 29        00160      ADD      HL,HL      ;SHIFT THREE BIT POSITIONS
C006 C39A0A    00170      JP       0A9AH      ;RETURN ARGUMENT
0000          00180      END
00000 TOTAL ERRORS

100 REM TEST PROGRAM FOR SHIFT ASSEMBLY LANGUAGE PROGRAM
200 POKE 16526,0
300 POKE 16527,192
400 INPUT "VALUE=";M
500 A=USR(M)
600 PRINT "ORIGINAL=";M,"SHIFTEO RESULT=";A
700 GOTO 400

```

**Figure 4-11. Assembly-Language/BASIC
Argument Passing**

Disk BASIC USR Calls

The procedure for interfacing assembly-language programs to Disk BASIC is much the same as that for Level II BASIC. The chief difference is that more than one USR call is permitted. The format of the Disk BASIC USR call is $X=USR_n(M)$ where n is 0 through 9, permitting ten separate USR calls. Each USR call is first defined by a DEFUSR function of the form $DEFUSR=\&HXXXX$ where XXXX is the address of the assembly-language subroutine in hexadecimal format.

The CALL to 0A7FH to pass an argument from BASIC to the assembly-language routine and the JP to 0A9AH to pass an argument back to BASIC is the same as in Level II BASIC. The sequence for the same assembly-language program as above is shown for Disk BASIC in Figure 4-12.

```

100 REM TEST PROGRAM FOR SHIFT ASSEMBLY-LANGUAGE PROGRAM
200 DEFUSRO=&HC000
300 INPUT "VALUE=";M
400 A=USRO(M)
500 PRINT "ORIGINAL=";M,"SHIFTEO RESULT=";A
600 GOTO 300

```

**Figure 4-12. Disk BASIC USR
Sequence**

Handling Multiple Arguments

Many times it is necessary to pass more than one argument between Level II or Disk BASIC and an assembly-language routine. Since the USR call permits you to pass only one 16-bit value, how can you pass multiple arguments? There are a number of different ways to accomplish this.

Packing Arguments

First of all, arguments may be **packed** into 16 bits. If, for example, an *x* and *y* screen position must be specified, then two 8-bit fields, one in H and one in L may be used. If you need four arguments and they can be held in four-bit values, then you can use 4 four-bit fields in HL.

Figure 4-13 shows the technique of storing data in a common memory area in an example that SETs a series of points in the assembly-language routine. The points are defined in a table starting at location 32700 and terminated with a -1.

C000	00100	ORG	0C000H	
	00110	;*****		
	00120	; ROUTINE TO PUT AN "0" IN OISPLAY.		
	00130	; ENTRY: BASIC HL LINKAGE POINTS TO TABLE OF ADDRESSES		
	00140	; EACH TABLE ENTRY IS SCREEN CHARACTER POSITION		
	00150	; TO BE SET. TABLE TERMINATED BY -1.		
	00160	;*****		
	00170	;		
C000	CD7F0A	SET0	CALL	0A7FH ;GET ADDRESS OF TABLE
C003	E5	00190	PUSH	HL ;NOW IN STACK
C004	DDE1	00200	POP	IX ;NOW IN IX
C006	DD6601	00210	SET010	LD H,(IX+1) ;MS BYTE
C009	006E00	00220	LO	L,(IX+0) ;LS BYTE
C00C	7C	00230	LD	A,H ;GET MS BYTE
C00D	FEFF	00240	CP	OFFH ;TEST FOR END
C00F	C8	00250	RET	Z ;GO IF DONE
C010	3E4F	00260	LO	A,'0' ;0 FOR STORE
C012	77	00270	LD	(HL),A ;STORE 0
C013	DD23	00280	INC	IX ;BUMP PNTR
C015	0023	00290	INC	IX ;TWICE
C017	18ED	00300	JR	SET010 ;CONTINUE
0000	00310	END		
00000	TOTAL ERRORS			

100	'BASIC DRIVER TO STORE OS IN OISPLAY
150	CLS
200	POKE 32700,32:'POKE 3E20H
300	POKE 32701,62
400	POKE 32702,34:'POKE 3E22H
500	POKE 32703,62
600	POKE 32704,36:'POKE 3E24H
700	POKE 32705,62
800	POKE 32706,255:'POKE -1
900	POKE 32707,255
1000	0EFUSRO=&HC000
1100	B=32700:'START OF POKED DATA
1200	A=USRO(D):'MAKE CALL TO AL ROUTINE
1300	GOTO 1300:'LOOP HERE TO RETAIN OISPLAY

32700	32	}	3E20H
1	62		
2	34	}	3E22H
3	62		
4	36	}	3E24H
5	62		
6	255	}	FFFFH
7	255		

Figure 4-13. Passing Multiple Arguments

Passing The Arguments

A second method is to pass the arguments between BASIC and assembly language by using a common memory area. Arguments can be POKEd into the memory area in BASIC or obtained from the area by PEEKs. The assembly-language code, of course, can easily load or store the values. You can define this common memory area as part of the assembly-language program area itself or in a convenient location outside of the assembly-language program.

Another way to pass the arguments is to use the contents of HL to define the address of a **parameter list**. Arguments can then be picked up or stored in the list by the assembly-language program. Of course, the address of the parameter list can be passed back from the assembly-language routine to BASIC. The parameter list may be established in BASIC by using a **dummy string** or **array**. You can then determine the address of the array or string by a VARPTR function in BASIC and pass it to the assembly-language routine.

In the next chapter we'll look at a special case of interfacing assembly-language routines: **embedded** machine-language code in BASIC programs.

Chapter 5

Embedded Machine Code in BASIC

In this chapter, we'll discuss special techniques for **embedding** assembly-language code in BASIC programs. By this technique, short assembly-language routines can be included as DATA statements or strings in the BASIC program. One advantage to this is the ability to store, load, and execute the BASIC and assembly-language code as one file. Additionally, you don't have to compute the addresses of the assembly-language routines manually and include them in the BASIC code — they are automatically found by the BASIC program itself.

The drawbacks to this method are: (1) it works best with short code segments that are **relocatable**; (2) it requires some conversion of the hexadecimal machine code values into decimal values to include in the BASIC code.

Relocatable Code

Before we show several methods for embedding machine code, let's first talk about **relocatability**. As it applies to embedding, "relocatability" does not have the same meaning here as when we used the term with the Disk Assembler and Loader. The Loader relocates code by adding a **relocation bias** to addresses in instructions in order to obtain the correct address for the instruction no matter where in memory the instructions are moved. However, here we're talking about instructions whose machine code form is constant no matter where in memory the instruction is. These instructions would **not** have the relocation bias added to them during the loading process.

As a rule of thumb, any instruction that does not contain a memory address is relocatable. For example,

```
LD      A,B
AND     07FH
LDIR
LD      A,(HL)
```

are all instruction types which have the same machine code form no matter where in memory they're located.

Instructions that contain addresses, however, may or may not be relocatable. If the address reference is to somewhere inside the program and if the program is moved around in memory, the address in the instruction **should** be constantly changing depending upon the area of memory in which the program is loaded, as shown in Figure 5-1.

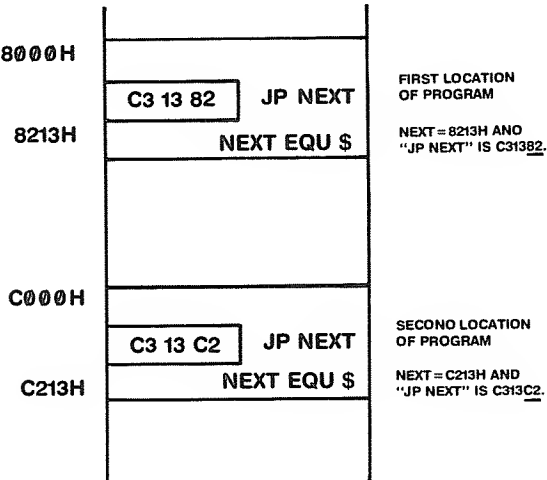


Figure 5-1. Non-Relocatable Direct Address Instruction

If the address in the instruction refers to a memory location that is not inside the program, such as a ROM subroutine or a fixed buffer in high memory, then even

though the program may move, the instruction containing the address is relocatable, as shown in Figure 5-2.

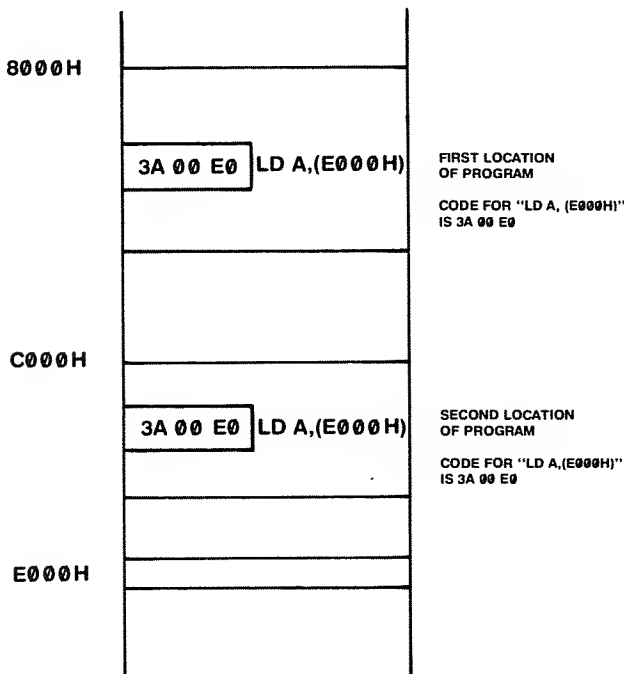


Figure 5-2. Relocatable Direct Address Instruction

Some examples of instructions that are **not** relocatable are

LD HL , TABLE (where TABLE is inside program)

LD A , (FLAGS) (where FLAGS is in program)

JP Z ,NEXT (where NEXT is inside program)

Jump instructions that jump to locations inside a program are always non-relocatable, unless they are "Jump Relative" instructions or DJNZ instructions. The reason for this (check out the instruction formats in Appendix II) is that the standard JP type instructions always have an address in bytes 2 and 3 of the instruction. This address, of course, should change as the program is moved around in memory.

JR and DJNZ Instructions

The format for the JR and DJNZ instructions is

$\begin{array}{|c|c|} \hline \text{BYTE 0} & \text{BYTE 1} \\ \hline \text{OF CODE 1} & \text{E-2} \end{array}$

DISPLACEMENT

the opcode is fixed:

JR=18H, JR C=1CH, JR NC=30H,

JR Z=28H, JR NZ=20H, DJNZ=10H

The second byte is computed by counting forward or back from the instruction following the JR or DJNZ. Since the JR or DJNZ is two bytes long, a jump to the following instruction would have a displacement in the second byte of 0; a jump to the JR or DJNZ itself would have a displacement of FEH or -2!

If you're weak on hex math, count back one byte at a time from the relative instruction like this: FDH, FC, FB, FA, F9, etc. to get the second byte. Best not to try this with an instruction 102 bytes back!

The JR and DJNZ instructions, however, use a **displacement** field of one byte in the instruction. This displacement field is used to compute the jump address by addition to the **current contents of the program counter**. Therefore, the jump address (or **effective address**) is always relative to the current location of the instruction. If the jump is to an instruction 6 bytes forward, the format of a JR NEXT is always 18 04; if the jump is to an instruction 6 bytes back, the format of a JR NEXT is always 18 F8. (Remember, the program counter points to the next instruction after the JR by the time it is added to the displacement, making the displacement field a value two bytes less than the displacement from the start of the JR!)

Because the JR and DJNZ are fixed values wherever the program is relocated in memory, they're relocatable instructions you can use in place of JPs. Of course, the jump address must be within 129 bytes forward or 126 bytes back of the JR or DJNZ.

If you're using the Disk Assembler, it automatically flags in the machine code section of the listing which instructions are relocatable and which instructions are non-relocatable. A byte that is relocatable is followed by a blank, while a single quote is used after an address. If the machine code for an instruction doesn't have a quote, the instruction is relocatable.

Hints and Kinks 5-2

Non-Relocatable Instructions

The following instructions are not relocatable unless they reference memory locations that are fixed and outside the program area in question:

Load A from memory	LD A,(nn)
Store A	LD (nn),A
Load register pair immediate	LD dd,nn dd=BC,DE,HL,SP,IX,IY
Load register pair memory	LD dd,(nn) dd=BC,DE,HL,SP,IX,IY
Store register pair	LD (nn),dd dd=BC,DE,HL,SP,IX,IY
Uncondi- tional jump	JP nn
Conditional jump	JP cc,nn cc=NZ,Z,NC,C,PO,PE,P,M
Uncondi- tional CALL	CALL nn
Conditional CALL	CALL cc,nn cc=NZ,Z,NC,C,PO,PE,P,M

Embedded Machine Code By DATA to Memory

The first method of embedding machine code is to include the code in DATA statements and then move it to a fixed location in memory. Let's see how this works on a simple program. Figure 5-3 shows the listing for a short program to scroll up the screen one line.

```

8000          00100      ORG      8000H
              00110 :SCROLL PROGRAM. SCROLLS UP SCREEN ONE LINE. FILLS
              00120 :LAST LINE WITH BLANKS
8000 21403C    00130 SCROLL LD      HL,3C00H+64      :START
8003 11003C    00140      LD      DE,3C00H          :DESTINATION
8006 01C003    00150      LD      BC,1024-64         :# BYTES
8009 EDB0      00160      LDIR                     :SCROLL
800B 21C03F    00170      LD      HL,3FC0H          :ADDRESS OF LAST LINE
800E 3E20      00180      LD      A,' '             :BLANK CHARACTER
8010 0640      00190      LD      B,64              :# BYTES PER LINE
8012 77        00200 LOOP LD      (HL),A            :STORE BLANK
8013 23        00210      INC      HL               :BUMP POINTER
8014 10FC      00220      DJNZ     LOOP              :GO IF NOT 54
8016 C9        00230      RET                        :RETURN
0000          00240      END
00000 TOTAL ERRORS

```

**Figure 5-3. SCROLL Program
Example**

The program contains all relocatable code and is meant to be called from BASIC by the techniques discussed in Chapter 4. To embed the code in a BASIC program by that method, take the following steps:

1. Assemble and debug the program completely.
2. Convert the hexadecimal values for the machine code into their decimal equivalents.
3. Put these decimal equivalents into a DATA statement in the BASIC program.
4. Move the DATA values to a predetermined area of memory at the beginning of the BASIC program.
5. Set up the USR address and call the machine code as required.

These steps have been carried out in Figure 5-4. The resulting BASIC program is a simple test of the program; two versions are shown, one for Level II and one for Disk BASIC. (Set MEMORY SIZE to 32767.)

Hints and Kinks 5-3
BASIC Address Computation

In using machine code in higher memory with BASIC—

Certain BASIC functions that operate only with 16-bit integer values don't permit "absolute" values from 32768 through 65535. An "overflow" error results. When addressing memory locations with these functions, the interpreter must be fooled into thinking the memory location value is a legitimate integer value from -32768 to +32767.

This problem only occurs for memory locations in the upper 32K of memory, 32768 through 65535. When you reference these locations, they must be in the form of the expression —

LOCATION-65536

This problem occurs for the FOR . . . TO command, DEFUSR command, PEEK command, POKE command, and others.

Example:

```
100 PRINT PEEK(60000) does not work,  
but  
100 PRINT PEEK(60000-65536) does.
```

```
100 REM EMBEDDED MACHINE CODE BY DATA TO MEMORY  
200 FOR I=32768 TO 32790  
300 READ A  
400 POKE (I-65536),A  
500 NEXT I  
600 DEFUSR0=&H8000: 'POKE 16526,0: 'FOR LEVEL II  
700 'POKE 16527,128: 'FOR LEVEL II  
800 X=USR0(0): 'X=USR(0): 'FOR LEVEL II  
900 GOTO 800  
1000 REM THESE DATA VALUES ARE THE 23 BYTES OF SCROLL IN DECIMAL  
1100 DATA 33,64,60,17,0,60,1,192,3,237,176,33,192,63,62,32,6,64  
1200 DATA 119,35,16,252,201
```

— DISK BASIC

— LEVEL II BASIC

**Figure 5-4. DATA-to-Memory
Embedded Machine Code**

In this case, relocatable code was used in the SCROLL program. However, since the predetermined area of 8000H was used, the code **could** have included non-relocatable instructions, as long as the assembly was performed with an ORG of 8000H.

This method can be used for any size of assembly-language program. It does get quite tedious, however, to convert the hexadecimal values into decimal for large amounts of code. A few other disadvantages include the necessity of fixing the memory area for the assembly-language code and the possibility of having to "merge" the DATA table with other DATA values in the BASIC program.

Embedded Machine Code By CHR\$ Strings

This method of embedding machine-language code into a BASIC program uses the CHR\$ function of Level II or Disk BASIC to construct a string of machine-language values. You can then find the location of the string by the VARPTR function and make a call by the USR function.

You must carry out the same preliminary steps as in the first method: the program must be assembled, debugged, and converted to decimal values. The decimal values must then be used in a BASIC character string such as ZZ\$=CHR\$(xx)+CHR\$(xx)+CHR\$(xx)+ . . . where "xx" represents the machine code decimal values.

A BASIC program that calls SCROLL and uses this method is shown in Figure 5-5. The VARPTR function is used to find the location of string ZZ\$. Remember from the Level II manual that VARPTR points to a parameter block for the string. The parameter block has the form shown in Figure 5-6. The second and third bytes represent the actual location of the string.

```

50 REM CHR$ EMBEDDED MACHINE CODE
100 A$=CHR$(33)+CHR$(64)+CHR$(60)+CHR$(17)+CHR$(0)+CHR$(60)
+CHR$(1)+CHR$(192)+CHR$(3)+CHR$(237)+CHR$(176)+CHR$(33)
+CHR$(192)+CHR$(63)+CHR$(62)+CHR$(32)+CHR$(6)+CHR$(64)+CHR$(119)
+CHR$(35)+CHR$(16)+CHR$(252)+CHR$(201)
200 B=VARPTR(A$)
300 C=PEEK(B+2)*256+PEEK(B+1):'POKE 16526,PEEK(B+1):'FOR LEVEL II
400 IF C>32767 THEN C=C-65536:'POKE 16527,PEEK(B+2):'FOR LEVEL II
500 DEFUSR0=C:'REM FOR LEVEL II'
600 X=USR0(0):'X=USR(0):'FOR LEVEL II
700 GOTO 600

```

--- DISK BASIC
 ——— LEVEL II BASIC

Figure 5-5. CHR\$ Embedded Machine Code

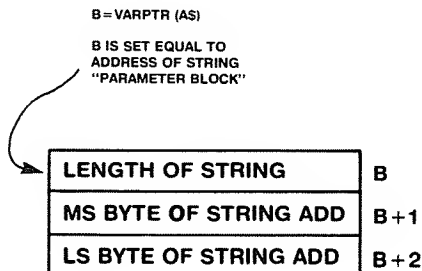


Figure 5-6. String Parameter Block

Because the string is located in the string storage area high in memory, there is a major problem with this method! Both Level II and Disk BASIC go through a "garbage collection" process when they run out of string space. As you recall, a string storage area may be CLEARED initially in the BASIC program.

As operations are performed on strings in the BASIC program, the interpreter uses more and more of the string storage area. The interpreter is **sloppy** and does not clean up old forms of the strings as new string storage space is

allocated. However, after many string operations, the interpreter may run out of string storage space and have to go back to "clean up" the string storage area. When this is done, the memory for the strings are reallocated and the string addresses change.

This garbage collection may never happen in the BASIC program you interface to the assembly-language code. It is a function of memory size, string storage area, and string operations. However, because it **can**, it is always best to find the address of the CHR\$ string **immediately before** the machine code in the string is called.

This method works quite well for short assembly-language programs, but one of the limitations is in the length of the BASIC line defining the string. To avoid the problem, several strings may be concatenated to produce a larger string, however, a more stringent limitation is the size of the string itself, which cannot be greater than 255 characters. You must use all relocatable code in this method.

DATA Values and Dummy Strings

A third way to embed assembly-language code in BASIC programs is to use a table of DATA values representing machine code as in the first method but to move those values to a "dummy" string. The advantage of this method over the first is that the BASIC program will perform the address calculation automatically by the VARPTR function. The disadvantages are the string size limitation and the requirement of all relocatable code.

The "dummy string" here is a BASIC string statement of the form ZZ\$="THIS IS A DUMMY STRING. . . ." The number of bytes in the string should be equal to or greater than the number of bytes in the assembly-language program. The text is irrelevant. The location of the string is found by VARPTR, and the bytes from the DATA table replace the characters of the dummy string.

In this case, reshuffling strings is no problem during the

garbage collection process since the string location is in the BASIC statement itself and the interpreter records the address of the text as the string address. Figure 5-7 illustrates use of this method with the SCROLL program.

Hints and Kinks 5-4

Using Dummy Strings

There is a slight problem in using a dummy string as a storage area for machine code. BASIC uses a zero (null) byte to mark the end of a text line. If a zero is stored for a byte of an instruction, the BASIC interpreter will truncate the line during edits, lists, and other operations.

You can store zeroes and use the embedded machine language normally. However, don't attempt to edit such a program after execution. Edit the BASIC program before it's run and save it on cassette or disk before execution.

If a program is listed after execution, strange things will occur as the interpreter encounters the modified dummy string. Valid machine codes may not be valid display characters!

```

100 REM DATA TO DUMMY STRING EMBEDDED MACHINE CODE
200 A$="THIS IS A DUMMY STRING!"
300 B=VARPTR(A$)
400 C=PEEK(B+2)*256+PEEK(B+1)
500 FOR I=C TO C+22
600 READ A
700 IF I>32767 THEN POKE I-65536,A ELSE POKE I,A
800 NEXT I
900 IF C>32767 THEN DEFUSR0=C-65536 ELSE DEFUSR0=C
1000 'POKE 16526,PEEK(B+1):'FOR LEVEL II
1100 'POKE 16527,PEEK(B+2):'FOR LEVEL II
1200 X=USR0(0):'X=USR(0):'FOR LEVEL II
1300 GOTO 1200
1400 DATA 33,64,60,17,0,60,1,192,3,237,176,33,192,63,62,32,6,64
1500 DATA 119,35,16,252,201

```

-- -- DISK BASIC

_____ LEVEL II BASIC

**Figure 5-7. DATA to Dummy String
Embedded Machine Code**

DATA Values and Array Storage

By taking advantage of the contiguous nature of arrays, you'll have a fourth method of storing machine code within the BASIC program. Arrays in BASIC are allocated by DIM statements. Once you've established the array, the array is treated as a block of data. Any number of bytes can be used in the array. The array location remains fixed as long as new variables are not referenced; its location can always be found by VARPTR.

— Hints and Kinks 5-5 — Array Storage of Machine Code

If an integer array (such as A%) is used for storage of machine code data, remember that the elements of the array are two bytes long and that the data is stored in normal array operations in standard Z-80 address format. Storing 0 through 4 in array A%(0)-A%(4), for example, produces

LOC+0	Ø	}	$A\%(0) = 0$
+1	Ø		
+2	1	}	$A\%(1) = 1$
+3	Ø		
+4	2	}	$A\%(2) = 2$
+5	Ø		
+6	3	}	$A\%(3) = 3$
+7	Ø		
+8	4	}	$A\%(4) = 4$
+9	Ø		

Take this into account when storing machine code data by setting an array element to two machine code bytes —

$A\%(4) = (\text{machine code byte } n) * 256 + (\text{machine code byte } n+1)$

In this method, the DATA values representing the machine code are moved to a dummy array, the VARPTR function is used to find the address of the array, and the DATA values are moved initially to the array. To make storage into the array more straightforward, the array should preferably be an array of integer values, since they occupy two bytes per element. The elements of an array are **contiguous**, that is, they occupy successive locations in memory, and the DATA values can be stored with no problem. Figure 5-8 shows this method.

```

100 REM DATA TO ARRAY EMBEDDED MACHINE CODE
200 DIM A%(12)
300 B=VARPTR(A%(0))
400 FOR I=0 TO 11
500 READ C,D
550 E=D*256+C
600 IF E>32767 THEN A%(I)=E-65536 ELSE A%(I)=E
700 NEXT I
800 IF B>32767 THEN DEFUSR0=B-65536 ELSE DEFUSR0=B
900 POKE 16526,B-INT(B/256):FOR LEVEL II
1000 POKE 16527,INT(B/256):FOR LEVEL II
1100 X=USR0(0):X=USR(0):FOR LEVEL II
1200 GOTO 1100
1300 DATA 33,64,60,17,0,60,1,192.3,237,176,33,192.63,62,32,6,64
1400 DATA 119,35,16,252,201,0

```

-- DISK BASIC
 ____ LEVEL II BASIC

NOTE DUMMY LAST VALUE FOR EVEN #

**Figure 5-8. DATA to Array
Embedded Machine Code**

Passing Arguments and Multiple Subroutines

In the above examples, we illustrated the techniques with a very simple subroutine. However, you can incorporate all of the material presented in Chapter 4, into these embedded machine-language methods including passing multiple arguments and multiple subroutines back and forth.

Using embedded machine-language code can be a valuable tool since it blends the easy-to-use string handling, display and print formatting, and number-crunching abilities of BASIC with the high-speed of assembly language. It's probably well worth your effort to take portions of BASIC programs that are notoriously slow and rewrite them into short embedded assembly-language routines — code such as sorts, searches, and graphics. Watch for further examples of embedded machine-language code in later chapters.

SECTION II

Assembly-Language Techniques

Chapter Six

Number Crunching

This chapter looks at what is commonly called “number crunching.” This term refers to operations on data that is primarily numeric — adds, subtracts, multiplies, divides, trigonometric functions, integration, differentiation, and so forth. We only have room here to discuss basic number-crunching operations that come up frequently in non-math oriented programs — such things as adds, subtracts, multiplies, divides, and random number generation.

Addition and Subtraction

Addition and subtraction operations are built into the instruction set of the Z-80. All other math-related operations have to be built out of adds, subtracts, compares, and shifts.

Eight-Bit Adds

The basic adds and subtracts use two 8-bit operands. One of the operands is in the A register, while the other operand is either in the instruction itself (immediate addressing), in another CPU register (register addressing), in a memory location pointed to by the HL register (register indirect addressing), or in a memory register pointed to by the **effective address** of an indexed-type instruction.

Suppose we have 23H in the A register, 45H in the B register, 8000H in the HL register pair, 8040H in IX, 7FF0H in IY, and 45H in memory location 8000H. All of these adds accomplish the same thing, adding a 45H to the 23H in the A register, putting the result of 68H in the A register and setting the flags.

- | | |
|-------------------|----------------------------------|
| 1. ADD A,45H | Immediate add of 45H to 23H in A |
| 2. ADD A,B | Add of 45H in B to 23H in A |
| 3. ADD A,(HL) | Add of 45H in 8000H to 23H in A |
| 4. ADD A,(IX-40H) | Add of 45H in 8000H to 23H in A |
| 5. ADD A,(IY+16) | Add of 45H in 8000H to 23H in A |

Generally, in **arithmetic** instructions, all flags are affected. But how are the flags set? The C flag is set when there is a carry from bit 7 of the result, which does not occur here. The Z flag is set when the result is zero, which is not the case for the result of 68H. The P/V flag is the dual-purpose flag used either for **parity** or **overflow**. In arithmetic instructions, P/V is always used for overflow. There is no overflow here, and P/V is not set. The sign flag is set when the result has a one bit in bit 7, indicating that the result is negative (not the case here). The N flag is not set here since this is an add, and the half-carry flag is not set here because there is no carry from bit 3 of the result. (N and H are hardly ever used by the programmer, and there are no conditional branches on them — so ignore them!)

P/V Flag

When we talk about parity in the TRS-80, we're not discussing farm prices. Parity refers to a count of the number of one bits in (usually) an 8-bit location or register. Parity is primarily used to check the data on I/O operations. Since I/O devices lose data more frequently than cpu/memory actions, a check on the number of one bits is a rough check on the validity of the data.

In the Z-80, the P/V flag is set for even parity. If the number of one bits is even after an instruction affecting parity, then the P/V flag is set; otherwise it is reset. Instructions that set parity in this fashion are IN R,(C), most shifts and rotates, and the logical instructions.

The P/V flag is used for overflow when (most) arithmetic instructions are executed. It can be tested by a conditional jump to ascertain whether or not the result was too large to be held as a two's complement number, either for 8-bit or (some) 16-bit instructions. Overflow occurs for 8 bits when a result is below -128 or above 127 and for 16 bits when a result is below -32768 or above 32767.

Many times an add will be used without any other following action. Sometimes, however, an add will be followed by a conditional branch of some type, as in the example from the MEMORY Subroutine from Chapter 14 shown in Figure 6-1. An ADD A,(HL) is done to add the contents of the memory location pointed to by HL to the contents of the A register. If the result is zero, a conditional jump is made to MEM008 to set the count to zero (!); if the result is positive (P), a JP is made to MEM010.

0013'	F1	00320	POP	AF	
0014'	F5	00330	PUSH	AF	
0015'	86	00340	ADD	A, (HL)	; ADJUST COUNT
0016'	CA 001C'	00350	JP	Z, MEM008	; GO IF ZERO
0019'	F2 001D'	00360	JP	P, MEM010	; GO IF POSITIVE
001C'	AF	00370	MEM008:	XOR	A
001D'	FE 64	00380	MEM010:	CP	100
001F'	FA 0024'	00390	JP	H, MEM020	; GO IF LT 100
0022'	3E 63	00400	LD	A, 99	; MAX COUNT
0024'	77	00410	MEM020:	LD	(HL), A
					; STORE COUNT

**Figure 6-1. Arithmetic Followed
by Conditional Branch**

Many times programmers get confused about using conditional branches on flags. Much of the confusion is about when the flags get set and reset. **Unless an instruction description specifically says that the flags are affected by the instruction, the flags remain as they are!** This means that you can use flags from several instructions back for conditional branching as long as you haven't used instructions that affect the flags **in question**. Many times, however, you will use a conditional JP as the next instruction after the arithmetic instruction.

Eight-Bit Subtracts and Compares

Eight-bit subtract instructions operate in the same addressing modes as the ADD instruction. The flags are set in almost identical fashion, with a few exceptions. The SUB sets the C flag if there is **no borrow**. This means, in fact, that the C flag is reset if there's a carry out of bit 7 — just the reverse of the ADD!

Carries

When is a carry set, and when is it reset on an add, subtract, or compare? Here's how you can find out manually: If an add is being performed, add the two operands in binary; any carry bit out of the high order will simply set the carry.

On a subtract or compare, convert the two operands to binary. Now take the two's complement of the subtrahend (the one to be subtracted). Now add the operands. Complement the state of the carry. The result will be the state of the carry after a subtract or compare. After running through several thousand test cases, we discovered that the carry was set on a subtract or compare if a larger unsigned number was subtracted from a smaller unsigned number. What about signed numbers? This will be left as an exercise for those readers who are not faint of heart

$ \begin{array}{r} 00000011 \quad 3 \\ 00000010 \quad -(+2) \\ \hline \text{CY} \quad 11111110 \quad \text{2'S COMP} \\ \downarrow \leftarrow 00000001 \quad +1 \text{ RESULT} \\ 0 \end{array} $	$ \begin{array}{r} 00000010 \quad 2 \\ 00000011 \quad -(+3) \\ \hline \text{CY} \quad 11111101 \quad \text{2'S COMP} \\ \downarrow \leftarrow 11111111 \quad -1 \text{ RESULT} \\ 0 \end{array} $
$ \begin{array}{r} 11111110 \quad 254 \\ 11111111 \quad -(+255) \\ \hline \text{CY} \quad 00000001 \quad \text{2'S COMP} \\ \downarrow \leftarrow 11111111 \quad -1 \text{ RESULT} \\ 1 \end{array} $	$ \begin{array}{r} 00000001 \quad 1 \\ 00000000 \quad -(+2) \\ \hline \text{CY} \quad 11111111 \quad \text{2'S COMP} \\ \downarrow \leftarrow 11111111 \quad -1 \text{ RESULT} \\ 1 \end{array} $

You would think that adding -5 to 10 in A would be the same as subtracting 5 from 10 in A. However, it isn't, as the C is set for the ADD and reset for the SUB. (Also, the N and H flag logic is different for the ADD and SUB.) The result in A is still the same.

```

LD   A,10    ;10 TO A
ADD  A,-5     ;10 PLUS -5 TO A
JP   C,OUT    ;THIS JUMP IS MADE
,
,
,
LD   A,10     ;10 TO A
SUB  A,5      ;10 MINUS 5 TO A
JP   C,OUT    ;THIS JUMP IS NOT MADE

```

Compares operate in identical fashion to subtracts, except that the result is thrown away and is only used to set or reset the flags. Figure 6-2 shows use of the SUB and CP from the DECBIN subroutine of Chapter 13.

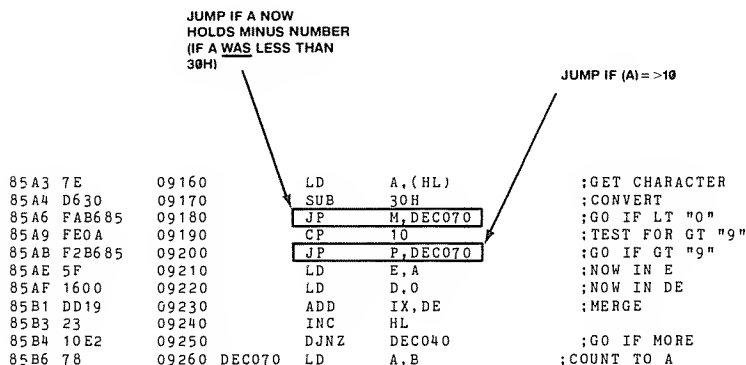


Figure 6-2. Use of SUB and CP

Sixteen-Bit Adds and Subtracts

Sixteen bit adds and subtracts can use either HL, IX, or IY as **16-bit accumulators** similar to the A register. However, the number of addressing modes that can be used for register pairs is limited to one, the one associated with adding another register pair to HL, IX, or IY. Also, while you can do an ADD or ADC to HL, IX, or IY, you aren't allowed a SUB or SBC for IX or IY.

Another difference in the 16-bit arithmetic instructions is that the flags **may** or **may not** be affected, according to the instruction. ADD HL,BC, for example, does not affect Z, P/V, or S, while ADC HL,BC **does** affect these flags. To see which flags are affected keep one eye on Appendix II while using these instructions with conditional JPs.

Figure 6-3 shows you a trick to use with the C flag. The ADD HL,DE does **not** affect the Z flag, but does affect the C flag. In this subroutine, the programmer did a decrement on the contents of HL by loading DE with -1 and performing the ADD HL,DE. The C flag will be set as long as HL is positive or zero, but will be reset when HL reaches -1.

00100	TITLE	DELAYS
00110	ENTRY	DELAY
00120	*****	
00130	; DELAY SUBROUTINE	
00140	; DELAYS 1 TO 65536 MILLISECONDS.	
00150	; ENTRY: (HL)=DELAY COUNT IN MILLISECONDS	
00160	; 0=65536	
00170	; ALL REGISTERS SAVED	
00180	*****	
00190	; SAVE REGISTERS	
00200	DELAY:	PUSH BC
00210		PUSH DE
00220		PUSH HL
00230		LD OE,-1
00240		DEC HL
00250	DELO10:	LD B,131
00260	DELO20:	LD HL,OE
00270		ADJ C,DELO10
00280		JP
00290		POP HL
00300		POP OE
00310		POP BC
00320		RET
00330		END

Figure 6-3. Using the C Flag for ADD HL,XX

Both the 8-bit and 16-bit adds and subtracts have instructions that add or subtract the carry (or borrow). ADC A,23H, for example, adds not only 23H to the contents of A, but also the current state of the carry. SBC HL,BC subtracts not only BC from HL, but also subtracts the carry.

These instructions are used primarily for **multiple-precision** operations involving more than 8 or 16 bits of data. The carry or borrow must be **propagated** from the lower order to the higher order byte or "word."

Hints and Kinks 6-3 Multiple-Precision Operations

How far can multiple-precision operations be carried out? It's entirely possible to implement arithmetic that works with 100 bytes of precision. The question is, is it necessary? To get a rough idea of the number of decimal digits that can be contained in any size binary number, take the number of bits and divide by 3.5 — 8 bits would be 2.3, 16 bits would be 4.5, 32 bits would be 9.1, and so forth (4.5, for example, would be somewhere between four and five decimal digits). You can see it doesn't take too many bits before we can express ten or twenty decimal digits of precision.

Aside from mathematical games or precise scientific applications, there's generally no need for large multiple-precision numbers. The compromise reached in all computers is to hold floating-point numbers with a dozen or so decimal digits precision and an exponent that represents a power of 16. This format would still call for multiple-precision operations on the "'fractional'" part, but it would be limited to operations involving approximately 3 to 6 bytes.

Even with a small number of bytes, multiple-precision multiplies and divides are tedious and slow. Rather than writing a bit-by-bit divide to work with 64 bits, many programmers would implement something like this: A four-byte number can be expressed as $AB \cdot 16^2 + CD$, where A, B, C, and D are the byte values. To multiply two such numbers in four-byte precision would involve the expansion —

$$ABCD * EFGH = AB * EF + CD * EF + AB * GH + CD * GH$$

This would require four 16-bit multiplies, considerable multiple-precision shifting, and four multiple-precision adds! This is a lot of work for assembly-language programmers, who are notorious for looking for the easy way out!

Figure 6-4 shows a use of this in the SUB subroutine from Chapter 13. SUB subtracts the contents of a four-byte variable SEED through SEED+3 from the contents of DE,HL. DE,HL are treated as a four-byte variable. Note that initially the carry is cleared by an OR A instruction. One of the peculiarities of the Z-80 is that it has a SCF to set the carry, and a CCF to **complement** the carry, but no Reset Carry; the OR A **does** reset the C flag, however, and the contents of A remain unchanged. The next SBC is not preceded by an OR A to reset the carry, and the carry from the first SBC is subtracted from the result.

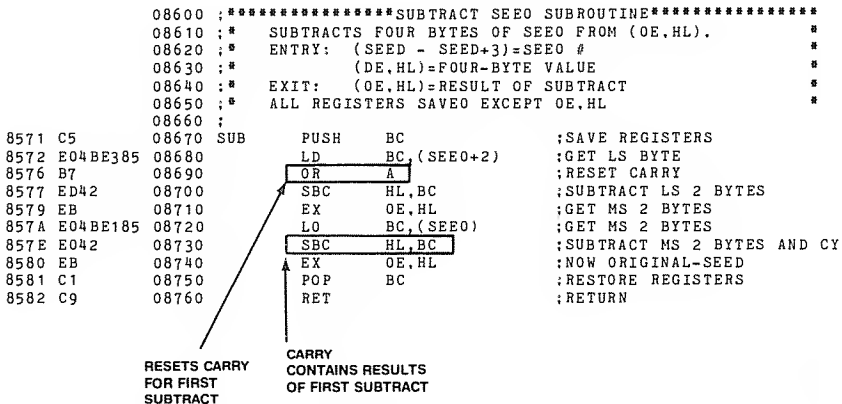


Figure 6-4. Four-Byte Subtract Example

Multiplies and Divides

All multiplies and divides on the Z-80 must be performed **in software**. There are a number of methods for performing software multiply and divides, ranging from successive addition and subtraction to high-speed **table lookups**. We'll discuss some of the more common ways here.

Software Multiplies

Successive Addition

The simplest multiply is by **successive addition**, as shown in Figure 6-5 from Chapter 13. Here, the DE register contains the multiplicand and is added to HL (after HL is cleared) a number of times corresponding to the multiplier. Three adds are used to multiply by three.

81C9	ED53E585	02560	LD	(DOT0),DE	;STORE DOT ON TIME
81CD	210000	02570	LD	HL,0	
81D0	19	02580	ADD	HL,DE	;FIND 3*DOT0
81D1	19	02590	ADD	HL,DE	
81D2	19	02600	ADD	HL,DE	
81D3	22E785	02610	LD	(DASH0),HL	;STORE DASH ON TIME

Figure 6-5. Multiply by Successive Addition

A variation on this approach uses the multiplier in the B register to take advantage of the DJNZ instruction.

```
MULT LD    B,10    ;MULTIPLIER =10
      LD    DE,34   ;ARBRITRARY MULTIPLICAND
      LD    HL,0    ;CLEAR RESULT
LOOP  ADD   HL,DE   ;SUCCESSIVE ADD
      DJNZ  LOOP    ;GO IF NOT N TIMES
```

Shift and Add Multiply

Another type of multiply is the “shift and add.” This method, as the name implies, is a combination of shifts and adds. You can factor any multiplier into a number of powers of two factors. You can multiply N by 20, for example, by adding $16*N + 4*N$. Sixteen and 4 are powers of 2 that you can easily obtain by shifting.

This type of multiply works best for commonly used multipliers, like 10. The code in Figure 6-6 below, excerpted from the Decimal to Binary Conversion subroutine of Chapter 13, shows how you can use this method. The number is shifted by an `ADD IX,IX`, which shifts the number one bit position left. Four shifts result in $N*8$. The previously saved $N*2$ shift is then “popped” into `DE` and added in to give $N*10$.

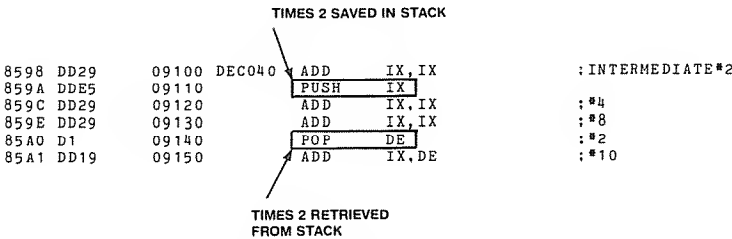


Figure 6-6. Shift and Add Example

In the multiplies above, you’ll have speed limitations in the successive addition case (for large multipliers, the process takes a very long time) and lack of generality in the shift and add case. For programs that perform many multiplies of different sizes of numbers, it’s best to implement a bit-by-bit multiplication.

Bit-By-Bit Multiply

A bit-by-bit multiply, implemented as a subroutine, is presented in Figure 6-7. The multiplier is held in `DE` on entry and the multiplicand is in `BC`. On exit, the result is

held in DE, HL, treated as a four-byte register. If the product of the multiply is less than 65536, then it is in HL alone with zero in DE.

```

8000          00050          ORG          8000H
00100 ;*****
00110 ;*          SWEET 16 MULTIPLY          *
00120 ;* ENTRY: (OE)=MULTIPLIER, 16 BITS UNSIGNED          *
00130 ;*          (BC)=MULTIPLICAND, 16 BITS UNSIGNED          *
00140 ;* EXIT: (DE,HL)=PRODUCT, 32 BITS UNSIGNED          *
00150 ;*          (BC)=RETAINED          *
00160 ;*          (A)=OE VASTATED          *
00170 ;*          EVERYTHING ELSE IN REASONABLE ORDER          *
00180 ;* HISTORICAL NOTES: THIS MULTIPLY WAS WRITTEN WHILE          *
00190 ;*          PULLING TOGETHER THE FINAL PIECES OF THIS BOOK          *
00200 ;*          IN ABOUT 23577 MILLISECONDS.(NO, NOT THE 1ST TIME). *
00210 ;*****
00220 ;
8000 210000 00230 SWE16 LD HL,0 ;ZERO PRODUCT HALF
8003 3E10 00240 LD A,16 ;ITERATION COUNT
8005 EB 00250 SWE010 EX DE,HL ;SETUP FOR SHIFT
8006 29 00260 ADO HL,HL ;SHIFT DE
8007 F5 00270 PUSH AF ;SAVE POSSIBLE CARRY
8008 EB 00280 EX DE,HL ;PUT OE BACK
8009 29 00290 ADO HL,HL ;NOW BOTTOM HALF
800A 3001 00300 JR NC,SWE020 ;GO IF NO CARRY
800C 13 00310 INC OE ;PROPOGATE
800D F1 00320 SWE020 POP AF ;GET C FROM DE
800E 3004 00330 JR NC,SWE030 ;GO IF NONE
8010 09 00340 ADD HL,BC ;BIT WAS A ONE
8011 3001 00342 JR NC,SWE030 ;GO IF NO CARRY TO MSB
8013 13 00344 INC OE ;PROPOGATE
8014 3D 00350 SWE030 DEC A ;DECREMENT COUNT
8015 20EE 00360 JR NZ,SWE010 ;GO IF MORE
8017 C9 00370 RET ;BACK TO CALLING PROG
0000 00380 ENO
00000 TOTAL ERRORS

```

Figure 6-7. Bit-by-Bit Multiply Code

The subroutine works by shifting DE, HL. The bit shifted out of DE is used to determine whether you should perform an add of BC to HL. HL holds the partial product. Sixteen shifts are performed, and each one causes either an add or no action dependent upon whether the carry is a one or zero. The action of the multiply is shown in Figure 6-8.

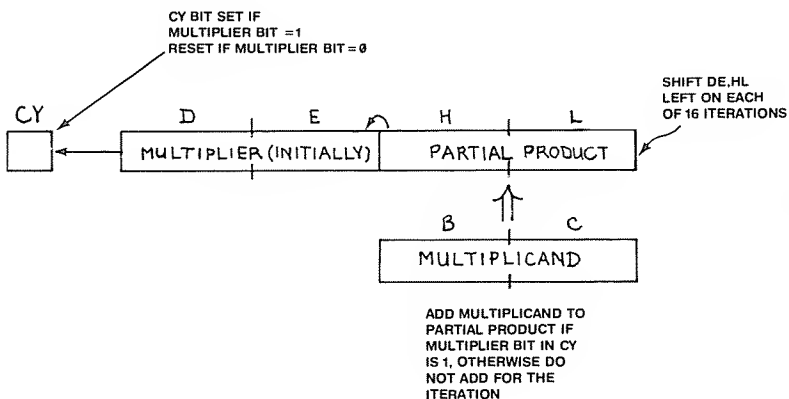
Fast Multiplies

It seems that computers are constantly being pushed by the applications! By that I mean that there's a constant requirement for faster and more efficient computers to handle applications that haven't been computerized and could be, or to speed up existing applications.

One of the critical areas of application involves multiplies as they are used in all kinds of number crunching. Since we don't have a hardware multiply on the TRS-80, we have to make do with a software routine.

You can speed up multiplies considerably by making them 'in-line' code. This method eliminates the usual 8- or 16-iteration loop by substituting 8 or 16 code segments. Multiplies of 150 microseconds for 8 bits are possible with this approach, a factor of two better than some iterative multiplies.

Other high-speed multiplies involve constructing large tables of partial results for various multiplier/multiplicand combinations.



**Figure 6-8. Bit-by-Bit Multiply
Action**

Software Divides

Successive Subtract Method

Software divides are handled in similar fashion to multiplies. Figure 6-9 shows a successive subtract method from the code of Chapter 13. It divides a 16-bit dividend in HL by a 16-bit divisor in BC. Each time the divisor is successfully subtracted from the dividend without producing a negative result (no carry), a count in DE is incremented by one. The count was initially set to -1. At the end of the divide, DE holds the count, which is the quotient. You could find the remainder by adding BC back to the **residue** in HL, although this is not done in the code.

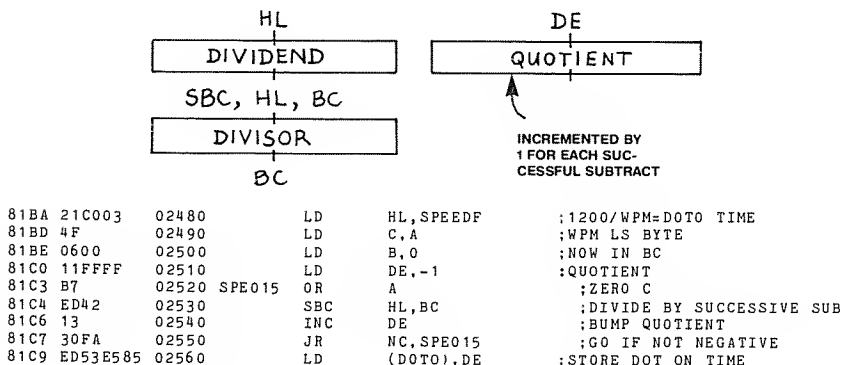


Figure 6-9. Divide by Successive Subtraction

Bit-By-Bit Restoring Method

Another type of divide is implemented in the code from Chapter 14 shown in Figure 6-10. This is a bit-by-bit “restoring” divide, which divides the 16-bit dividend in HL by an 8-bit divisor in E. On exit, HL holds the quotient and BC holds a remainder. The action of the divide is shown in Figure 6-11.

```

00100      TITLE  OIVISR
00110      ENTRY  OIVIDE
00120      ;*****
00130      ;* PERFORMS AN UNSIGNED OIVIDE OF 16 BITS BY 8-BIT
00140      ;* DIVIDE SUBROUTINE
00150      ;* OIVISOR.
00160      ;* ENTRY:  (HL)=DIVIDENO
00170      ;*          (E)=OIVISOR
00180      ;* EXIT:   (HL)=QUOTIENT
00190      ;*          (BC)=REMAINDER
00200      ;* ALL REGISTERS SAVED EXCEPT HL, BC
00210      ;*****
00220      ;
00230      DIVIDE: PUSH  OE
00240      ;          PUSH  IX
00250      ;          PUSH  HL
00260      ;          POP   IX
00270      ;          LD    HL, 0
00280      ;          LD    O, 0
00290      ;          LD    B, 16
00300      DIVO10: A00  HL, HL
00310      ;          A00  IX, IX
00320      ;          JR    NC, DIVO20
00330      ;          INC  HL
00340      DIVO20: INC  IX
00350      ;          OR   A
00360      ;          SEC
00370      ;          JR    NC, DIVO30
00380      ;          A00  HL, DE
00390      DIVO30: DEC  IX
00400      ;          DJNZ DJNZ
00410      ;          PUSH HL
00420      ;          POP  BC
00430      ;          PUSH IX
00440      ;          POP  HL
00450      ;          POP  IX
00460      ;          POP  DE
00470      ;          RET
00480      ;          END
;*****
;SAVE REGISTERS
;DIVIDENO TO IX
;ZERO HL
;DIVISOR IN DE
;ITERATION COUNT
;SHIFT 16 MS BITS
;SHIFT 16 LS BITS
;GO IF NO CARRY
;CARRY TO 16 MS BITS
;SET Q BIT
;CLEAR CARRY
;TRY SUBTRACT
;GO IF DIVIDE WENT
;RESTORE
;RESET Q BIT
;GO IF NOT 16 ITERATIONS
;REMAINDER
;QUOTIENT
;RESTORE REGISTERS
;RETURN

```

Figure 6-10. Bit-by-Bit Divide Routine

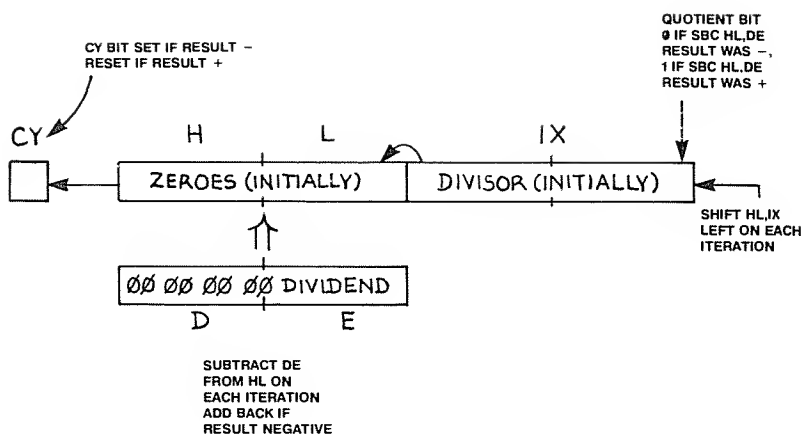


Figure 6-11. Bit-by-Bit Divide Action

Signed Vs. Unsigned Multiplies and Divides

All of the multiplies and divides we've discussed above were **unsigned**. That is, all operands were considered to be positive numbers with no sign bits. This means that the range of numbers held in a 16-bit result would be 0 through 65535, rather than -32768 through +32767.

You may wonder if a signed multiply or divide is possible. Yes, but the rules for a signed operation are somewhat sticky. It is best to find the **absolute value** of the operands, perform the unsigned multiply and divide, and then convert the result to the proper sign.

Overflow Limits

Another problem that we haven't discussed in the above code is **overflow**. Overflow may occur when the register(s) dedicated to the product or quotient is not large enough to hold the largest possible case.

For an unsigned multiply, overflow can occur if the results registers do not hold at least a number of bytes equal to the total number of bytes in the multiplicand and multiplier. For example, a multiply of a one-byte number by a two-byte number may produce a three-byte result (but not greater than three bytes).

For an unsigned divide, overflow may occur if the register(s) dedicated to the quotient is not equal to or greater than the number of bytes in the dividend. The worst case here is the divide-by-one case. Divide-by-zero is not a legitimate operation and will result in a quotient of all ones.

How do you detect overflow? Not by the P/V flag, which is usually unrelated to the multiply or divide actions! You must know the ranges of the numbers you'll be dealing with beforehand or be certain by testing the operands that no multiply or divide operation will cause overflow.

Random Number Generation

Many times it's necessary to generate "random" numbers. The uses for random numbers ranges from simple applications such as determining whether a plague will strike in "Hammurabi," to more complex simulation and modeling.

There are really two types of "random" numbers — pseudo-random numbers and true random numbers. True random numbers aren't predictable, but pseudo-random numbers are.

True Random Numbers

An example of a true random number generation occurs when a (perfect) die is rolled or when a (perfect) coin is flipped. The next toss of the die or flip of the coin is completely unpredictable. Over the long run the

distribution of the points coming up on the die will approach an even number of ones, twos, threes, fours, fives, and sixes, and there will be close to an equal number of heads and tails for the coin flip.

Can we get a true random number on the TRS-80? The answer is a qualified yes. One way to get a 7-bit random number is to read the contents of the R register in the Z-80. The R register refreshes the dynamic memories by counting from 0 to 127 and then recycling. If an external event occurs at irregular intervals that are very large compared to the R register counting, a true random number is obtained.

One such external event is a keypress. If we look at R at every keypress, we can obtain a good random number from 0 through 127. The code for such an operation looks for a keypress and then reads R into the A register.

```
READ CALL INPUT      ;READ KEYBOARD
      JR      Z,READ  ;GO IF NO KEYPRESS
      LD      A,R      ;NOW HAVE RANDOM
                        # 0-127 IN A
```

Pseudo-Random Numbers

A pseudo-random number is predictable. As a matter of fact, it's convenient to have code that will start from a given "seed" value and generate a whole string of numbers without repeating each time we start from the same seed. If we have such a routine, we can still get a good distribution of all numbers over the range and repeat the sequence any time we wish.

— Hints and Kinks 6-5 —
Random Number Problems

One of the main rules in random number generation is this: Never use any old algorithm to generate what you think will be a string of random numbers. If you take the contents of HL, multiply by your Social Security number, and add your best 10 kilometer running time to produce a series of pseudo-random numbers, you're almost guaranteed of producing a series that "decays" down to 18766 after 32 numbers have been produced. It's best to consult a good programming text for some tried and true algorithms. Even there, you'll find vehement disagreement among computer scientists about which methods are the best. (Personally, I take my wife's Social Security number)

One of the common algorithms for generating random numbers is to multiply an odd power of five times the seed value. The multiply produces a 16-bit or greater product (whatever we care to make it). When overflow results, the product represents the remainder of a 64K (for 16 bits), with the divide due to the length of the register. This divide is called a **modulus** operation. The product is the next pseudo-random number, and the process is repeated indefinitely. The greater the length of the register holding the product, the longer the **cycle** before the pseudo-random number sequence repeats. For 32 bits, the cycle is millions of numbers.

Two pseudo-random sequences are used in the codes of Chapters 13 and 14. They're basically the same. One is shown in Figure 6-12.

Figure 6-12.
Pseudo-Random Number
Routine

The RAND routine performs a shift and **subtract** of the four-byte SEED value. Seven shifts of SEED in DE, HL are done by calling the SHIFT subroutine. This multiplies the old seed by 128. Then the old SEED is subtracted three times from the shift result to give SEED*125. The new seed is stored into SEED for the generation of the next pseudo-random number.

Towards Infinite Precision

A frequent question in assembly language is how to handle large numbers. **Multiple-precision** adds and subtracts are relatively easy to implement (by using ADC and SBC instructions), but multiplies and divides of many bytes are tedious to code.

When the range of numbers is very great, a better approach is to implement **floating-point** processing in the assembly language. Floating-point holds numbers in a scientific-notation format geared to microcomputers (see Figure 6-13). Unfortunately, the implementation of this code for such processing is fairly complex.

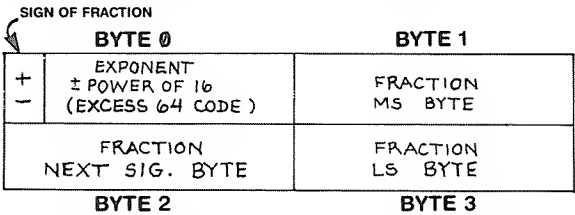


Figure 6-13. Floating-Point Number Representation (Typical)

Disk Editor/Assembler users have the **arithmetic routines** of the Disk Editor/Assembler library available to them to perform addition, subtraction, multiplication, division, and exponentiation of both **integer** and **real (floating-point)** numbers. The Disk Editor/Assembler manual describes the ways to use these options, but their use basically involves setting up arguments for the function registers and then referencing the library routine by a CALL. The library routine is then automatically loaded at **load time** by the Disk Editor/Assembler Loader.

Chapter Seven

Working With Character Data

Assembly-language routines to generate and process ASCII character data are our topics for this chapter. The first of these routines is the procedure for reading ASCII characters from the TRS-80 keyboard. Although ROM routines can be used, constructing your own keyboard **scanning** and conversion routines are not difficult and offer a lot of versatility.

Other common character-processing routines found in assembly-language programs are string input routines to read in user character strings, message output routines to display messages on the screen, character output routines to echo input characters, and routines to convert between binary data and ASCII **decimal** strings.

We'll get to all of these routines in this chapter, paying close attention to the **keyboard scanning** process and conversion to ASCII.

Keyboard Operation and Scanning

There are built-in routines in Level II BASIC and Disk BASIC to read the keyboard and convert a keypress into a character. The addresses and calls are documented in other Radio Shack literature. However, we can bypass these and read the keyboard directly. In doing so, we can make the routine as versatile as we want by making the keys represent any character or function!

The keyboard is really a **matrix** of switches as shown in Figure 7-1. There's nothing magical about its operation. Pressing a key simply connects a column line and a row

line together for as long as the key is held down. There's a certain amount of **keybounce** associated with the connection. If we could observe the connection from a microscopic scale in slow motion, we'd see a definite "make-break-make . . ." before the connection was firmly established. The same thing would happen when the key was released. This process is shown in Figure 7-2.

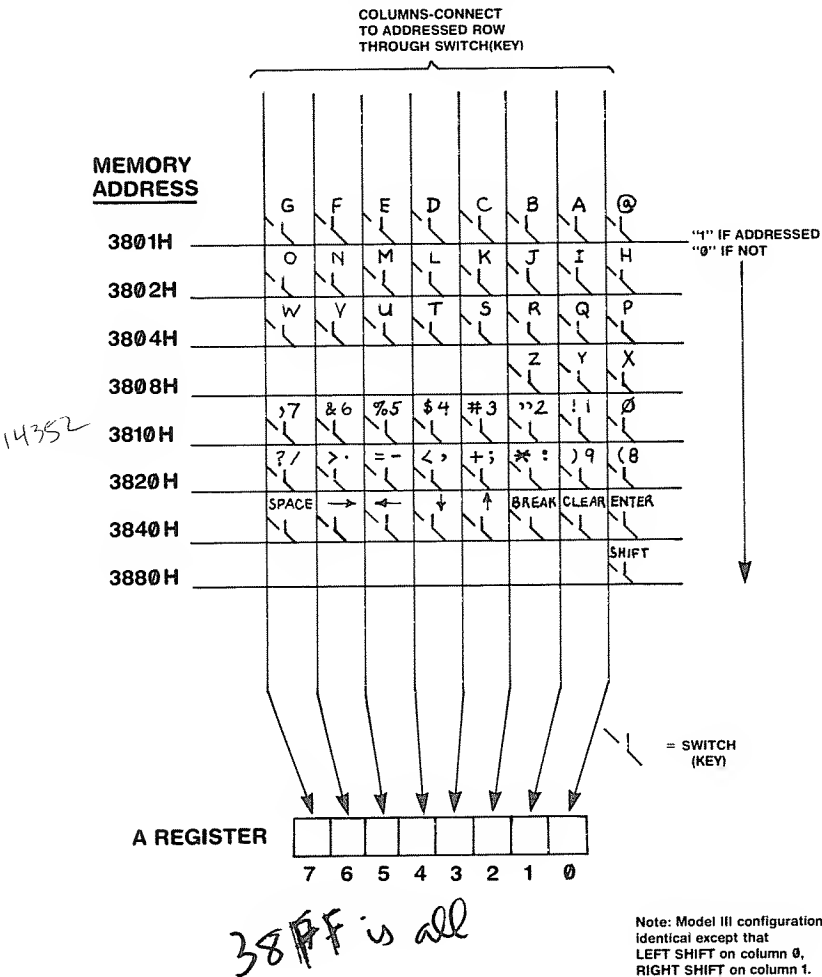


Figure 7-1. TRS-80 Model I
Keyboard Configuration
(Upper Case)

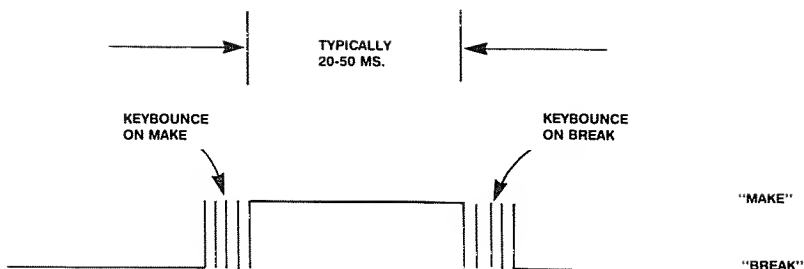


Figure 7-2. Keyboard Bounce

Our job, and it's fairly simple, is to convert a switch connection defined by a row, column into an ASCII character and to **debounce** the contact to avoid reading the same key many times.

Scanning

Each of the eight rows is addressed by a unique address, as shown in Figure 7-1. The address of the first row is 3801H, the second 3802H, the third 3804H, the fourth 3808H, the fifth 3810H, the sixth 3820H, the seventh 3840H, and the last 3880H. How do we address the rows? By simply doing a "load" from the memory location representing the row. To address the third row, we'd do

```
READ LD A,(3804H) ;READ KEYBOARD ROW 2
```

Here, we've called the row "row 2", as we counted from "row 0".

When the LD is executed, the A register is loaded with a byte that represents the state of the column lines at that instant in time. A column line will hold a one bit **if a key along the addressed row is being held down**. If more than one key along that row is being held down, there will be several one bits. Note that the keys associated with other rows are not detected at all.

The process of addressing eight rows, one at a time, and looking for a one bit representing a keypress is called **keyboard scanning**. Since an LD instruction takes about 8 microseconds (8 millionths of a second), you can do this keyboard scan very quickly relative to a keypress. You might hold down the typical key for 50 milliseconds or so, representing the time of 250 complete scans!

The processing for detecting any keypress, then, is really very simple and goes something like this:

1. Read row 0, address 3801H. Look for a non-zero value.
If zero, go on else go to step 9.
2. Read row 1, address 3802H. Look for a non-zero value.
If zero, go on else go to step 9.
3. Read row 2, address 3804H. Look for a non-zero value.
If zero, go on else go to step 9.
- .
- .
- .
7. Read row 6, address 3840H. Look for a non-zero value.
If zero, go on else go to step 9.
8. Scan complete and no key press. Go back to step 1.
9. Keypress here. Row number is known. Find column number by finding which bit of the eight columns is a one. Read the state of row 7, the SHIFT key and record it if necessary. Convert the key to ASCII or another code and take action on it.

Conversion

When the keyboard scan detects a non-zero value in any row, the next step to perform is conversion of the row, column representation into ASCII or some similar code. For example, the intersection of row 2, column (bit) 3 is associated with the key marked S on the keyboard.

The usual conversion of this key will, of course, be to an ASCII S, or 53H. There's no reason, however, that you can't convert this key to a code that means "scroll up," "insert,"

“get the next disk sector,” or anything else you want. The point is that the conversion routine simply converts one of 53 keys to any value the programmer wants. Usually these values are ASCII.

The **algorithm** for the conversion is this:

1. Take the least significant byte of the row value — 1,2,4,8,16,32, or 64. Convert to the row number of 0,1,2,3,4,5, or 6.
2. Multiply the row number by 8. We now have 0,8,16, 24,32,40, or 48.
3. Add in the column bit number of 0 through 7. We now have a unique value from 0 through 55, representing the key that has been pressed.
4. If the SHIFT key is to be used, add 0 if no SHIFT or 56 if shift. We now have a unique value of 0 through 111 representing the key that has been pressed and the “upper-” or “lower-case” status.
5. Use the value obtained above to get one of 56 or 112 bytes from a keyboard conversion table. The byte obtained will be the ASCII value of the key, or whatever code we want to use for the key.

— Hints and Kinks 7-1 —
Keyboard Algorithm

The index to the keyboard conversion table is given by:

$$\text{Index} = (\text{Log (base 2) (Row Address-3800H)}) * 8 + \text{Column Number} + 56 * (\text{Shift})$$

Here Shift=0 for no shift and 1 for shift

Suppose that the \$ key was pressed in Figure 7-1. We'd have:

$$\text{Index} = (\text{Log (base 2) (3810H-3800H)}) * 8 + \text{Column Number} + 56 * (\text{Shift})$$

$$\text{Index} = (\text{Log (base 2) (16)}) * 8 + 4 + 56 * 1 = 4 * 8 + 4 + 56 = 92$$

Debouncing

Debouncing is necessary because of the high speed of keyboard scanning. If we scanned the keyboard, converted a key, stored the value, and then came back to repeat the process for the next key, we'd probably be able to start the next scan after 200 microseconds. Since the typical key is held down for 50 milliseconds, we'd pick up the key 250 times again!

We need to **delay** a certain amount of time after we first detect and process the key. This delay will be long enough so that the key is released in the interim. If we are speaking of average typing speeds of 40 words per minute, the number of characters per second are about 4 or less. This is one every 250 milliseconds or so. Delaying 100 milliseconds then, should more than handle average typing speeds and yet bypass the period during which the key is held down.

Hints and Kinks 7-2

'Auto' Key

One of the easier things to implement in an assembly-language keyboard routine is an 'automatic' key function. If you hold the key longer than the 100 millisecond delay, it is reread. This results in an 'auto' key function for any key, which will operate at 10 characters/second.

There are more sophisticated keyboard decoding routines that handle "n-key rollover" in which you can press a new key while you are still holding the old, but the delay technique above is fine for most processing.

A Typical Keyboard Subroutine

We now have all the elements we need to read and convert the keyboard. Figure 7-3 shows a complete subroutine that will perform the task.

```

F000      00100      ORG      OF000H
00110      ; *****
00120      ; *      READ KEYBOARD ROUTINE      *
00130      ; *      ENTRY: NO PARAMETERS      *
00140      ; *      EXIT: (A)=KEY CONVERTED TO KBLUT VALUE      *
00150      ; *      OR ZERO IF NO KEY HAS BEEN PRESSED      *
00160      ; *****
00170      ;
00180      ; SCAN SEVEN ROWS
F000 210138      00190 READKB LD      HL,3801H      ;ROW 0 ADDRESS
F003 7E      00200 REA010 LD      A,(HL)      ;GET ROW VALUE
F004 B7      00210      OR      A      ;TEST FOR ZERO
F005 2005      00220      JR      NZ,REA020      ;GO IF KEY THERE
F007 CB25      00230      SLA      L      ;SHIFT ROW ADDRESS
F009 F8      00240      RET      M      ;GO IF LAST ROW, NO PRESS
F00A 18F7      00250      JR      REA010      ;MORE ROWS TO GO
      00260      ; CONVERT ROW, COLUMN TO INDEX
F00C 4F      00270 REA020 LD      C,A      ;ROW VALUE
F00D AF      00280      XOR      A      ;ZERO A
F00E CB3D      00290 REA025 SRL      L      ;SHIFT ADDRESS
F010 3804      00300      JR      C,REA035      ;GO IF DONE
F012 C608      00310 REA030 ADD      A,8      ;ROW*8
F014 18F8      00320      JR      REA025      ;CONTINUE
F016 06FF      00330 REA035 LD      B,0FFH      ;COLUMN COUNT
F018 04      00340 REA040 INC      B      ;BUMP COUNT
F019 CB39      00350      SRL      C      ;SHIFT ROW VALUE
F01B 30FB      00360      JR      NC,REA040      ;CONTINUE TILL "1" OUT
F01D 80      00370      ADD      A,B      ;NOW ROW*8+COL IN A
F01E 4F      00380      LD      C,A      ;TRANSFER TO C
      00390      ; FIND TABLE ENTRY
F01F 3A8038      00400      LD      A,(3880H)      ;SHIFT ROW ADDRESS
F022 B7      00410      OR      A      ;TEST FOR SHIFT
F023 2802      00420      JR      Z,REA045      ;GO IF NONE
F025 3E38      00430      LD      A,56
F027 81      00440 REA045 ADD      A,C      ;ADD HASH+SHIFT*56
F028 4F      00450      LD      C,A      ;NOW ROW*8+COL*SHIFT*56
F029 0600      00460      LD      B,0      ;NOW IN BC
F02B 213AF0      00470      LD      HL,KBLUT      ;ADDRESS OF LOOK UP TABLE
F02E 09      00480      ADD      HL,BC      ;POINT TO CODE
F02F 7E      00490      LD      A,(HL)      ;GET VALUE
      00500      ; DEBOUNCE DELAY
F030 210021      00510      LD      HL,8448      ;100 MS DELAY
F033 01FFFF      00520      LD      BC,-1      ;DECREMENT
F036 09      00530 REA050 ADD      HL,BC      ;DECREMENT COUNT
F037 38FD      00540      JR      C,REA050      ;GO IF NOT DONE
F039 C9      00550      RET      ;RETURN WITH CODE IN A
      00560      ; LOOK UP TABLE
0008      00570 KBLUT DEFS      8      ;ROW 0 LC
0008      00580      DEFS      8      ;ROW 1 LC
0008      00590      DEFS      8      ;ROW 2 LC
0008      00600      DEFS      8      ;ROW 3 LC
0008      00610      DEFS      8      ;ROW 4 LC
0008      00620      DEFS      8      ;ROW 5 LC
0008      00630      DEFS      8      ;ROW 6 LC
0008      00640      DEFS      8      ;ROW 0 UC
0008      00650      DEFS      8      ;ROW 1 UC
0008      00660      DEFS      8      ;ROW 2 UC
0008      00670      DEFS      8      ;ROW 3 UC
0008      00680      DEFS      8      ;ROW 4 UC
0008      00690      DEFS      8      ;ROW 5 UC
0008      00700      DEFS      8      ;ROW 6 UC
0000      00710      END
00000 TOTAL ERRORS

```

Figure 7-3.
Keyboard Read
Routine

The subroutine is divided into scanning, converting to an index value, finding the **lookup table** value, and debounce delay.

The scanning cycles through the addresses of 3801H through 3840H. HL is initialized with 3801H. Thereafter, only L is shifted left to get the "02H", "04H", etc. For each address, an LD A,(HL) is done to read the row. If the result is zero, the scan continues. If all addresses through 3840H are used and no key is found, the subroutine exits with zero in A.

If a non-zero value is found, the row address is converted to 8 times the row number by successive adds of 8. The column number is then added to this by simply adding the value obtained from the read. We now have 0 through 55.

The shift key is read by a LD A,(3880H). If there's a shift, a value of 56 is added to the value from above. If there's no shift, 0 is added.

The index value is then used to pick up one byte from the KBLUT, the keyboard look up table. The last action is to delay 100 milliseconds by a timing loop.

The KBLUT should be assembled with standard Radio Shack ASCII and **control codes**. Any other codes could be substituted by the user. The orientation of the codes is opposite that of Figure 7-1. The first 8 bytes, for example, would be the codes for @, A, B, C, D, E, F, and G.

Other Keyboard Subroutines

We used some variations on the above routine in the programs of Chapters 13 and 14. Chapter 13 uses a special keyboard routine that is geared to a quick scan and return if no key has been pushed. The quick scan is implemented by an LD A,(387FH) shown in Figure 7-4. This reads in **all** of the rows at one time and **merges** all column bits. If there is a zero in A after this instruction, no key is being pressed. If the result is non-zero, the usual row-by-row scan is used.

```

06470 ;*****KEYBOARD INPUT SUBROUTINE*****
06480 ; * IF OEBOUNCE DELAY LESS THAN ELAPSEO TIME, SCANS *
06490 ; * KEYBOARD AND STORES POSSIBLE INPUT CHARACTER IN *
06500 ; * CIRCULAR INPUT BUFFER. *
06510 ; * ENTRY: NO PARAMETERS *
06520 ; * EXIT: NO PARAMETERS *
06530 ; * ALL REGISTERS SAVED *
06540 ; * CLEAR CHARACTER CAUSES RESTART AT MOR015H, SP RESET *
06550 ;
8429 F5 06560 INPUT PUSH AF ;SAVE REGISTERS
842A 3A7F38 06570 LD A,(387FH) ;ALL IN ONE SWELL FOOP
8420 B7 06580 OR A ;TEST FOR ANY KEY
842E CAB984 06590 JP Z,INP065 ;GO IF NONE
8431 C5 06600 PUSH BC
8432 E5 06610 PUSH HL
8433 2AE085 06620 LO HL,(TSLC) ;GET TIME SINCE LAST CHARACTER
8436 016400 06630 LO BC,OBOEL ;MINIMUM OELAY

```

READS ALL
KEYBOARD ROWS,
MERGES ALL
COLUMN BITS

Figure 7-4. Quick Scan

The INPUT subroutine of Chapter 14 (Figure 7-5) is a specialized keyboard input. It looks only for a 0 through 8 key by reading row 4 and row 5. Row 4 addresses keys 0 through 7, while row 5 addresses only key 8 in bit 0. If a key is found, a 100 millisecond delay is done by calling the DELAY subroutine.

```

00100 TITLE INPUTSR
00110 INPUT
00120 EXT DELAY
00130 *****
00140 INPUT SUBROUTINE *****
00150 ** SCANS KEYBOARD FOR INPUT OF 0,1,2,3,4,5,6,7, OR
00160 ** 8 KEYS. IGNORES ALL OTHERS. WAITS FOR KEY PRESS.
00170 **
00180 ** ENTRY: NO PARAMETERS
00190 ** EXIT: (A)=0,1,2,3,4,5,6,7, OR 8 IN BINARY
00200 ** ALL REGISTERS EXCEPT A SAVED
00210 *****
00220 INPUT: PUSH BC ;SAVE REGISTERS
00230 PUSH HL
00240 INF002: LD A,(3810H) ;KEYBOARD ROW 4
00250 OR A ;TEST FOR NON-ZERO
00260 JR Z,INF010 ;GO IF NON
00270 LD C,0FEH ;COLUMN COUNT
00280 INC C ;BUMP COUNT
00290 HRAA ;SHIFT OUT
00300 JR NC,INF005 ;GO IF NOT CARRY
00310 LD A,C ;COUNT TO A
00320 JR INF020 ;CONTINUE
00330 INF010: LD A,(3820H) ;KEYBOARD ROW 5
00340 AND 1 ;GET "8" KEY ONLY
00350 JR Z,INF002 ;GO IF NONE
00360 LD A,8 ;FOR 8 KEY
00370 INF020: LD HL,100 ;100 MILLISEC DELAY
00380 DELAY
00390 POP HL ;RESTORE REGISTERS
00400 POP BC
00410 RET
00420 END

```

READ 8
LOGIC

READ
8-7
LOGIC

Figure 7-5. Special Keyboard Read

Input Subroutines

Once you have implemented a keyboard input routine, it's fairly easy to write code to read in a string of characters. This is usually a subroutine that reads in the string but doesn't do **syntax checking** or make any judgments about the validity of the data that has been read in; that task is left to other routines that will process the input character string.

The "input string" subroutine must, however, make some checks. First, there must be some terminating input character that signifies the end of the input process. If there were not, the input-string subroutine would call the keyboard-input subroutine again for the next character, when the user was actually through with input. The terminating character in the TRS-80 is almost always ENTER.

Second, there may be a maximum number of characters to be input. You may need to specify the number because the **input buffer** is of limited size, or because all input should be less than a certain number of characters.

Other options for the input string routine might include a means to **backspace** (rubout) an incorrect character or characters from the string and a returned count of the number of characters input.

— Hints and Kinks 7-3 —

Backspacing

Backspacing is not implemented in the input routines of Chapters 14 and 15. The reason is that most input in routines of those chapters involves either one or two characters which are easily redone if an error is made.

Also, implementation of backspacing is best done when the input involves an entire line of data, as it does in BASIC. When a line is input, an input buffer is filled with the character data before any processing is done. Processing is done after the terminating character (ENTER in BASIC) has been entered. When the "input line" approach is used, it is a simple matter to detect a backspace (left arrow or rubout), adjust the buffer pointer to point back at the last character, and then overwrite the last character.

Have I given enough excuses?

Figure 7-6 shows a somewhat atypical input string subroutine from the MORG program of Chapter 13. It calls a keyboard input routine called INPUTW that returns the next keyboard character. INPUTW specifically waits until a key has been pressed and **always** returns a character. The INPUTS subroutine terminates on an ENTER character, and each new character is compared to ENTER, which is EQU(ated) earlier in the program. (Note here that the code for ENTER in this keyboard program is 02H.) If the character input is an ENTER, the subroutine is exited. Otherwise a character count is incremented in the C register, and the character is displayed on the screen by the DISCHR subroutine.

```

06180 ;*****INPUT STRING SUBROUTINE*****
06190 ; INPUTS STRING OF CHARACTERS AT CURRENT COMMUNICA- *
06200 ; TION AREA. TERMINATED BY ENTER. *
06210 ; ENTRY: (B)=MAXIMUM NUMBER *
06220 ; (CURCUR)=CURRENT CURSOR POSITION *
06230 ; EXIT: (B)=ACTUAL NUMBER INPUT *
06240 ; (HL)=FIRST CHARACTER LOCATION *
06250 ; NZ IF GT MAXIMUM NUMBER *
06260 ; Z IF LE MAXIMUM NUMBER *
06270 ; ALL REGISTERS SAVED EXCEPT HL,BC,A *
06280 ;
8400 2AE985 06290 INPUTS LO HL,(CURCUR) ;CURRENT CURSOR POSITION
8410 E5 06300 PUSH HL ;SAVE
8411 04 06310 INC B ;BUMP MAXIMUM
8412 0E00 06320 LO C,0 ;INITIALIZE COUNT OF CHARS
8414 C5 06330 INS010 PUSH BC ;SAVE COUNTS
8415 CDF384 06340 CALL INPUTW ;GET CHARACTER
8418 C1 06350 POP BC ;RESTORE COUNTS
8419 FE02 06360 CP ENTER ;TEST FOR 00NE
841B 2809 06370 JR Z,INS030 ;GO IF ENTER
8410 0C 06380 INC C ;BUMP CHARACTER COUNT
841E C0BE83 06390 CALL OISCHR ;OISPLAY
8421 10F1 06400 OJNZ INS010 ;GO IF NOT MAXIMUM
8423 3EFF 06410 LO A,OFFH ;-1 TO A
8425 B7 06420 OR A ;RESET Z FLAG
8426 E1 06430 INS030 POP HL ;RETRIEVE START
8427 41 06440 LO B,C ;GET CHARACTER COUNT
8428 C9 06450 RET ;RETURN
06460 ;

```

Figure 7-6. Input String Routine

The usual keyboard input routine is a `'read-one-character-and-wait-until-input'` type. It will never return until that character has been input. The rationale is that this is the most frequent type of input; the program asks for user input before processing.

The scanning type of input routine is used less frequently. Here, the program is processing merrily along but is keeping an eye out for a user stimulus. This stimulus is usually a user `'abort'` action, although it could be normal keyboard input, as in the case of the MORG program of Chapter 13. The goal in this type of input routine is to make certain that the polling of the keyboard input is done periodically at regular measured intervals (so that no characters are missed) and yet done fast enough that other processing can continue without being overburdened by the keyboard poll.

In many other systems, input of infrequent (compared to CPU processing) keyboard characters would be accomplished by an interrupt from the keyboard when the next character was ready. That way, normal processing would continue until the keyboard interrupt was received, eliminating the polling task. One way to time the polling on a disk system is by using the system's real-time-clock. It provides 25 millisecond increments, which lend themselves nicely to timing any type of polling function.

The B register initially held the maximum number of characters to be input plus one. After each character is input, B is decremented by a DJNZ. If less than the maximum has been input, the code goes to the next

character. If the maximum number of characters has been input, the subroutine is terminated with an error flag of "NZ".

The characters are stored in the video display memory. This causes no problem as long as the characters are valid ASCII characters. However, non-standard characters may be changed in the upper-case versions of the TRS-80. The DISCHR routine is called to display the input characters because input in assembly language does not automatically display the characters as a BASIC input does!

Display of Characters

This brings us to a discussion of the next topic — what has to be done to display character data in assembly language? BASIC has built-in print driver routines that handle all output of characters. Character strings are "formatted" and tabbed, new lines are output, the screen "scrolls" automatically, etc.

Unless you make calls to the BASIC routines to perform these functions, you must implement all **display processing** via assembly-language routines. Fortunately, display processing is not that involved, and we'll look at some of the techniques here.

Displaying A Message

To display a message on the screen, a string of ASCII characters from a DEFM is output one at a time to the screen memory area. It's convenient to have a terminating character to end the message. The terminator we've used in the programs of Chapters 13 and 14 is a zero (null) at the end of each message. We used the zero because it is not a valid ASCII character and we can easily test it.

An alternative to using zero is to use the disk Editor/Assembler "DC" pseudo-op, which sets the high-order bit of the **last** character to one. This can also be easily detected (and masked out) for the "end of message."

DC Pseudo-Op

Perhaps the DC pseudo-op should have been used in the code of Chapters 13 and 14 since it is specifically for generating character strings for output. The output routine would then test the sign bit of every character and terminate when a 'minus' was detected. The scheme used in the code of the applications programs wastes one byte for each message. Thank goodness memory is inexpensive!

The screen area for the message is usually pointed to by the HL register or another register pair. It is incremented by one for every new ASCII byte output. Figure 7-7 shows the DSPMES subroutine from Chapter 13. This subroutine uses HL to point to the ASCII message and BC to address the "starting screen address." The starting screen address would be 3C00H through 3FFFH.

```

D4850 ;*****DISPLAY MESSAGE AT LOCATION N*****
0486D ;*   DISPLAYS MESSAGE AT GIVEN SCREEN POSITION. TER-   *
D4870 ;*   MINATES ON NULL (ZERO).                           *
0488D ;*   ENTRY: (HL)=MESSAGE LOCATION                       *
0489D ;*               (BC)=SCREEN POSITION                     *
049DD ;*   ALL REGISTERS SAVED.                                *
04910 ;
836D F5      D492D DSPMES   PUSH    AF                      ;SAVE REGISTERS
836E C5      D493D        PUSH    BC
836F E5      D494D        PUSH    HL
8370 7E      D495D DSPDD5  LD      A,(HL)                  ;GET MESSAGE CHAR
8371 B7      D496D        OR      A                        ;TEST FOR 0
8372 28D9    J497D        JR      Z,DSPD10                 ;RETURN IF DONE
8374 02      D498D        LD      (BC),A                   ;STORE CHARACTER
8375 03      D499D        INC     BC                       ;BUMP SCREEN POINTER
8376 23      D50DD        INC     HL                       ;BUMP MESSAGE POINTER
8377 ED43E985 D5D1D        LD      (CURCUR),BC            ;SAVE POINTER
837B 18F3    D5D2D        JR      DSP005                   ;CONTINUE
837D E1      D5D3D DSPD1D  POP     HL                      ;RESTORE REGISTERS
837E C1      D504D        POP     BC
837F F1      D5D5D        POP     AF
8380 C9      D5D6D        RET                               ;RETURN

```

Figure 7-7. Display Message Routine

DSPMES also stores the "current cursor position" in variable CURCUR. This variable can be tested for "end of line" or "end of screen" (scrolling) conditions.

Displaying an Input Character

When you input a character from a keyboard routine, it must be immediately "echoed" to the screen by a "display character" routine. This subroutine would typically take a "current cursor position" and use it as the address of the screen position to store the input character. Figure 7-8 shows such a routine that uses variable CURCUR as the current cursor position.

```

05510 ;*****DISPLAY CHARACTER SUBROUTINE*****
05520 ;*   OUTPUTS ONE CHARACTER TO CURRENT CURSOR POSITION   *
05530 ;*   ON SCREEN. MOVES CURSOR TO NEXT POSITION UNLESS  *
05540 ;*   LAST CHARACTER POSITION OF LINE 11. IF LATTER.   *
05550 ;*   SCROLLS UP FIRST.                                *
05560 ;*   ENTRY. (CURCUR)=CURRENT CURSOR POSITION           *
05570 ;*   (A)=CHARACTER TO BE OUTPUT                     *
05580 ;*   ALL REGISTERS SAVED.                             *
05590 ;
83BE C5 05600 DISCHR PUSH BC ;SAVE REGISTERS
83BF E5 05610 PUSH HL
83C0 2AE985 05620 LD HL,(CURCUR) ;GET CHARACTER POSITION
83C3 77 05630 (HL),A ;STORE CHARACTER
83C4 01FF3E 05640 LD BC,3C00H+767 ;LAST CP OF LINE 11
83C7 23 05650 INC HL ;BUMP CURSOR
83C8 22E985 05660 LD (CURCUR),HL ;STORE
83CB B7 05670 OR A ;RESET CARRY
83CC ED42 05680 SBC HL,BC ;TEST FOR LAST
83CE 2003 05690 JR NZ,DIS010 ;RETURN IF NO SCROLL
83D0 CDD683 05700 CALL SCROLL ;SCROLL UP
83D3 E1 05710 DIS010 POP HL ;RESTORE REGISTERS
83D4 C1 05720 POP BC
83D5 C9 05730 RET ;RETURN

```

Figure 7-8. Display Character Routine

Scrolling

DISCHR also tests CURCUR for a condition where **scrolling** is required. This condition normally occurs when the last character position on the screen, location 3FFFH, has been used. At that point you must scroll up the screen by moving lines 1 through 15 into lines 0 through 14, and "blank" the last line.

It's possible to scroll the first five or ten lines since we can format the screen any way we want — it's simply a matter

of coding! The SCROLL routine in Figure 7-9 scrolls the first 12 lines on the screen when the CURCUR reaches the end of the 12th line. Figure 7-10 shows the action. Scrolling on the TRS-80 is easy because of the LDIR block-move instruction and the fact that the screen is **memory mapped**. FILLCH is a "Fill Character" routine to fill the last line of the screen with blanks.

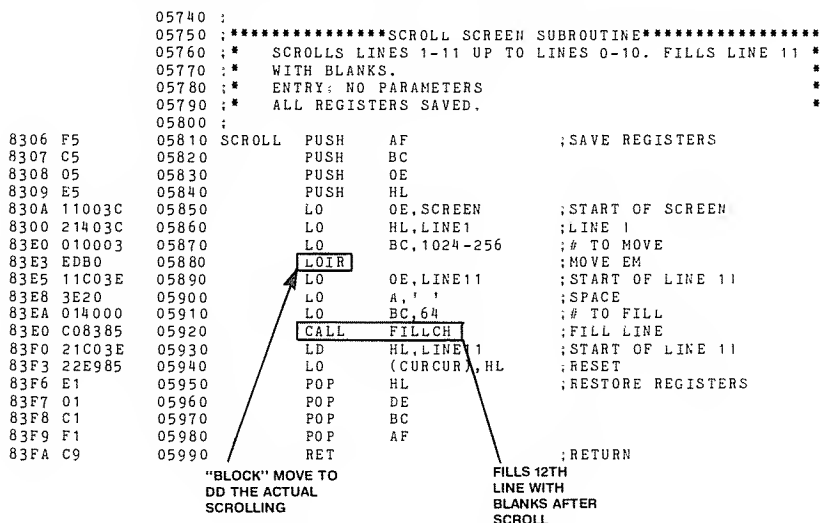


Figure 7-9. Scroll Screen Routine

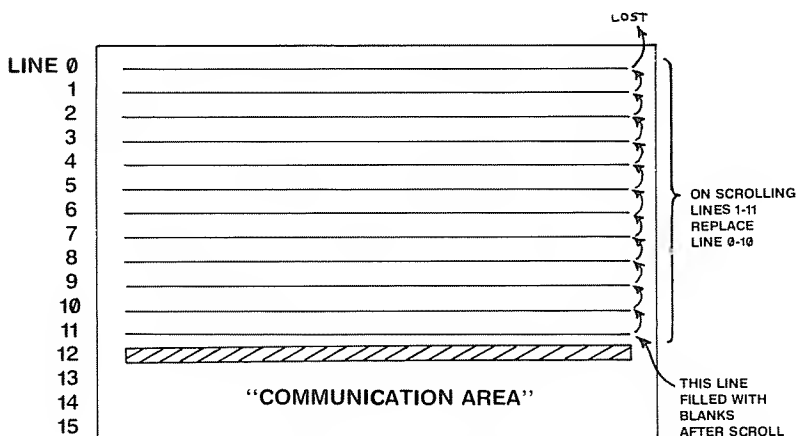
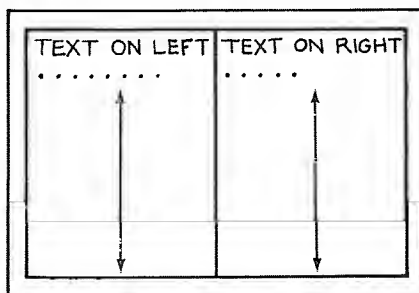


Figure 7-10. Scroll Action

Hints and Kinks 7-6

Multiple Scrolling

The TRS-80 screen can be divided up into any number of separate scroll areas by using the techniques shown in Figures 7-9 and 7-10. Implementing a 'split screen' capability, however, is another problem. Split screens split the screen into a right and left hand portion and can be very useful for comparing old text with new text or working with two 'documents' at a time.



One way to implement a split screen of this type would be to look for a screen address with the last 5 bits equal to 11111. This would denote the 31st character of either the left or right hand segments. This scheme works because lines start at '40H boundaries,' as they are 64 characters wide; line addresses always start at XX00H, XX40H, XX80H, or XXC0H. When this address was detected, the left or right hand pointer could be reset to the beginning of the line by adding 20H.

Scrolling is also more tedious (read 'messy'), because the areas involved are not contiguous. A scroll would have to be handled one 32-character line at a time.

All of this is possible, but it might be best to orient your applications towards multiple screens stacked on top of each other, rather than side by side - unless you're the stubborn type

Conversion From ASCII Decimal to Binary

A character string input from a keyboard or “input string” routine usually contains a mix of string data such as names and addresses and numeric data. The string data can be left as is in ASCII form for storage in arrays or records. However, you have to convert the numeric data to binary for processing. Any program that works with numeric data that is user entered invariably has a “decimal-to-binary” conversion routine.

The algorithm for converting from ASCII characters of 0 through 9 to binary goes something like this (and is shown in Figure 7-11):

1. Clear a result total.
2. Multiply the result total by 10.
3. Get the leftmost ASCII character.
4. Subtract 30H to get a binary value of 0 through 9.
5. Add the value to the result total.
6. Repeat steps 2 through 5 for the next leftmost character until done.

NUMBER IN BUFFER (ASCII)					
1 2 3 4 5					
STEP	RESULT	RESULT * 10	NEXT CHAR	- 30H	ADD TO RESULT
1	0	0	31H	1	1
2	1	10	32H	2	12
3	12	120	33H	3	123
4	123	1230	34H	4	1234
5	1234	12340	35H	5	12345
(BINARY)					

Figure 7-11. Decimal-to-Binary Action

You can carry out this conversion process for a string of ASCII digits as long as need be. Practical sizes, however, are limited to values that can be held in 16 or 32 bits. Since 16 bits can hold up to 65535, a decimal-to-binary routine that works in 16 bits can process most applications.

Decimal-to-binary routines also usually do testing of the ASCII characters to determine that they are valid ASCII values of 30H (0) through 39H (9). If they are found to be invalid values, the conversion is terminated with an "error" flag.

Figure 7-12 shows a DECBIN subroutine from Chapter 13 that converts a string of ASCII characters representing a decimal value to a binary value of 0 through 65535. A "shift and add" technique is used to multiply by 10. The routine is entered with HL pointing to the buffer containing the string and B containing the number of characters to be converted.

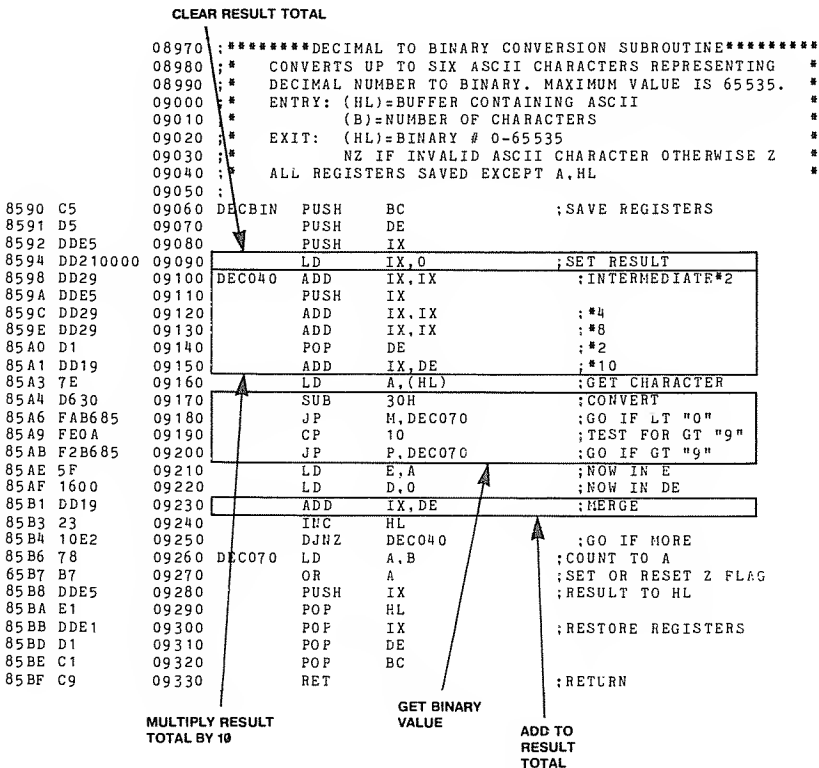


Figure 7-12. Decimal-to-Binary Routine

The conversion loop is executed “B” times. After the subtract of 30H, an error return is made if the result is negative (less than 30H), or greater than 9 (greater than 39H). An NZ condition is present on return if there was an invalid character.

Converting From Binary to Decimal ASCII

This processing converts binary data back into displayable or printable form. As in the case of DECBIN above, the same size limitations apply. Usually 16 or 32 bits of binary data are converted to ASCII decimal digits.

The “normal” algorithm for this conversion is the following:

1. Divide the binary data by 10.
 2. Save the quotient of the divide for the next divide.
 3. Add 30H to the remainder to produce an ASCII 0 through 9.
 4. Store the ASCII character at the next **rightmost** character position in a buffer.
 5. Repeat steps 1 through 4 until the quotient is zero.
- These steps are shown in Figure 7-13.

NUMBER IN REGISTER (BINARY)										
1 2, 3 4 5										
STEP	DIVIDE BY 10	Q	R	ADD 30H TO R	STORE IN BUFFER					
1	12345/10	1234	5	35H	<table><tr><td></td><td></td><td></td><td></td><td>5</td></tr></table> (ASCII)					5
				5						
2	1234/10	123	4	34H	<table><tr><td></td><td></td><td></td><td>4</td><td>5</td></tr></table>				4	5
			4	5						
3	123/10	12	3	33H	<table><tr><td></td><td></td><td>3</td><td>4</td><td>5</td></tr></table>			3	4	5
		3	4	5						
4	12/10	1	2	32H	<table><tr><td></td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>		2	3	4	5
	2	3	4	5						
5	1/10	0	1	31H	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr></table>	1	2	3	4	5
1	2	3	4	5						

Figure 7-13. Binary-to-Decimal Action

The BINDEC subroutine of Figure 7-14, however, employs an alternative approach. It uses a "divide by powers of 10" method of conversion. Starting with 10000, it performs successive subtractions on the binary number to divide by the power of 10. The quotient for each subtract is the power of 10 digit, which is then converted to ASCII. Successively smaller powers of 10 down to 1 are used. The algorithm for this is shown in Figure 7-15.

```

8000          00100          CRG          8000H
00110 ;*****
00120 ;          BINARY TO DECIMAL SR          *
00130 ; ENTRY:(HL)=16-BIT BINARY VALUE          *
00140 ;          (IX)=POINTER TO START OF CHARACTER BUFFER          *
00150 ; EXIT: (BUFFER)=FILLED WITH FIVE ASCII CHARACTERS,          *
00160 ;          LEADING ZEROES          *
00170 ;          (IX)=BUFFER+5          *
00180 ;*****
00190 ;
8000 FD212580 00200 BIN0EC LD          IY,PTABLE          ;POWER OF 10 TABLE
8004 AF        00210 BIN010 XOR         A          ;DIGIT COUNT TO 0
8005 F05601    00220          LO          0,(IY+1)          ;GET MS BYTE
8006 FD5E00    00230          LD          E,(IY+0)          ;GET LS BYTE
800B B7        00240 BIN020 OR          A          ;CLEAR CARRY
800C E052      00250          SBC         HL,OE          ;SUBTRACT POWER OF 10
800E 3803      00260          JR          C,BIN030          ;GO IF NEGATIVE
8010 3C        00270          INC         A          ;BUMP DIGIT COUNT
8011 18F8      00280          JR          BIN020          ;CONTINUE
8013 19        00290 BIN030 AOD         HL,OE          ;RESTORE TO POSITIVE
8014 C630      00300          AOD         A,30H          ;CONVERT TO ASCII
8016 007700    00310          LO          0,(IX+0),A          ;STORE IN BUFFER
8019 0023      00320          INC         IX          ;BUMP BUF POINTER
801E F023      00330          INC         IY          ;BUMP PWR 10 PNTR
801F FD23      00335          INC         IY
801F 7B        00340          LO          A,E          ;GET LS BYTE
8020 FE01      00350          CP          1          ;TEST FOR 5 DIGITS
8022 20E0      00360          JR          NZ,BIN010          ;GO IF NOT 5
8024 C9        00370          RET          ;RETURN
8025 1027      00380 PTABLE  OEFW       10000
8027 E803      00390          DEFW       1000
8029 6400      00400          OEFW       100
802B 0A00      00410          OEFW       10
8020 0100      00420          OEFW       1
0000          00430          ENO
00000 TOTAL ERRORS

```

Figure 7-14. Binary-to-Decimal Routine

NUMBER: 12345	POWER OF 10				
	<u>10 000</u>	<u>1000</u>	<u>100</u>	<u>10</u>	<u>1</u>
<u>-10 000</u>					
2345					
<u>-10 000</u>	<u>once</u>				
-XXXX					
<u>+10 000</u>					
2345					
<u>-1000</u>		<u>once</u>			
1345					
<u>-1000</u>		<u>twice</u>			
345					
<u>-1000</u>					
-XXXX					
<u>+1000</u>					
345					
<u>-100</u>			<u>once</u>		
245					
<u>-100</u>			<u>twice</u>		
145					
<u>-100</u>			<u>three</u>		
45					
<u>-100</u>					
-XXX					
<u>+100</u>					
45					
<u>-10</u>				<u>once</u>	
35					
<u>-10</u>				<u>twice</u>	
25					
<u>-10</u>				<u>three</u>	
15					
<u>-10</u>				<u>four</u>	
5					
<u>-10</u>					
-X					
<u>+10</u>					
5					
<u>-1</u>				<u>once</u>	
4					
<u>-1</u>				<u>twice</u>	
3					
<u>-1</u>				<u>three</u>	
2					
<u>-1</u>				<u>four</u>	
1					
<u>-1</u>				<u>five</u>	
0					
<u>-1</u>					
-X					
<u>+1</u>					
0					

Figure 7-15. "Powers of Ten"
Binary to Decimal

This subroutine converts a 16-bit binary number into 5 ASCII digits and returns a pointer to the last digit plus one. There are no error conditions for conversion of the 16-bit value.

—Hints and Kinks 7-7—

Converting Between ASCII and Binary or Hex

Converting from an ASCII string of characters representing binary or hexadecimal digits or the other way around isn't done as frequently as decimal/ASCII conversion. However, each conversion is relatively easy, since no multiplication must be done; the process simply involves stripping off one or four bits and translating them one at a time to ASCII, or vice versa.

To convert from ASCII to binary, get the ASCII character and subtract 30H. You now have a binary one or zero. Shift to the next bit position and repeat the process for the number of characters in the string.

To convert from binary to ASCII, get the next bit and add 30H. You now have an ASCII 0 or 1. Store in the output buffer and go on to the next bit.

To convert from ASCII to hexadecimal, do this: Get the next ASCII character. This will be 30H-39H, or 41H(A)-46H(F). Subtract 30H. If the result is greater than 9, subtract 7 (this adjusts for the ASCII characters between 9 and 0). You now have a hex 0 through F. Store in the next 4 bits and repeat the process for the next ASCII character.

To convert from hexadecimal to ASCII: Get the next group of four bits. Add 30H. If the result is greater than 39H, add 7. You now have an ASCII 0 through 9, A, B, C, D, E or F. Store in the buffer, and go on to the next group of four bits.

Chapter Eight

Working With Tables

Tables are some of the most common **data structures** used in assembly-language programming. Tables are so common that large programs that use many tables to define program operations are said to be **table driven** as opposed to more “processing-oriented.” We’ll discuss types of tables followed by some searching and sorting operations that can be performed in assembly language.

What are Tables?

A loose definition of a table is any collection of data items arranged in one contiguous block of memory. The data items may be **ordered** by some **key** value or **unordered**. Each **entry** in a table may consist of one or more bytes. The entry may contain a number of **fields** that are associated with the entry, or there may only be one field or item. The entry itself may be **fixed length** or **variable** length, and the table may also be either fixed or variable length.

Fixed Length Entry/Fixed Length Table

Let’s take a look at a typical simple table. The MOVETB from Chapter 15 is a five-entry fixed-length table; it’s shown in Figure 8-1. Each entry is two bytes long, making the total table length 10 bytes. Each entry of the MOVETB is an address defining the location of a “space cell” in a tic-tac-toe game (see Chapter 15).

MOVE TB + 0	
+1	ADDRESS FOR MOVE 1
+2	ADDRESS FOR MOVE 2
+3	
+4	ADDRESS FOR MOVE 3
+5	
+6	ADDRESS FOR MOVE 4
+7	
+8	ALWAYS 0
+9	

**Figure 8-1. Fixed-Length Entry/
Fixed-Length Table**

The entries in MOVETB are **indexed** by the move number of the tic-tac-toe game. The (computer) moves are numbered 1 through 4, so to get the address associated with any move, the formula is

$$\begin{aligned} \text{MOVETB ENTRY ADDRESS} &= \\ \text{MOVETB} + 2 * (\text{MOVE \#} - 1) \end{aligned}$$

Typical code for accessing the table is shown in Figure 8-2 from the MAIN3 subroutine of Chapter 15. Here, A is loaded with the move number, MOVENO. This is decremented by one to find the previous move number and then multiplied by two. This **index value** is then loaded into the BC register pair. IX is then loaded with the address of the table start MOVETB, and BC is added to IX. IX now points to the MOVETB entry for the previous move. The address in the entry is picked up in HL by two LD instructions that use the index register.

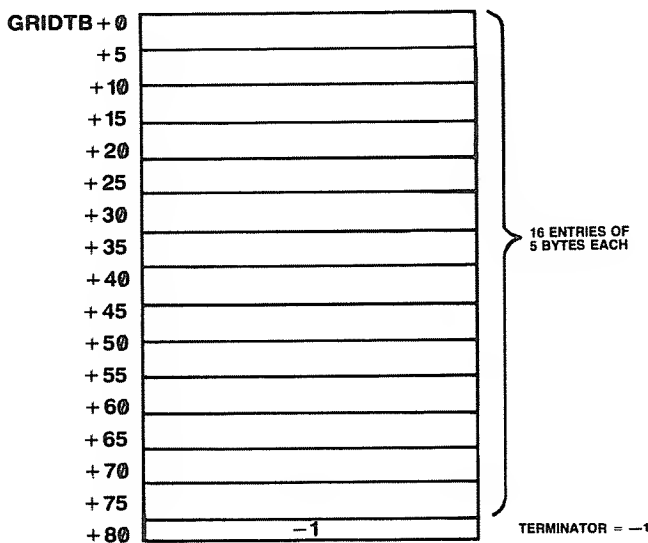
0DB5*	3A 00D0*	01240	; ALL SPACE CELLS DF THIS ENTRY ARE 0. REMOVE THIS
D0B8*	3D	D125D	; ENTRY BY ZEROING PREVIOUS LINK AND COCDEDE.
00B9*	07	0126D	LD A,(MDVENO)
00BA*	AF	0127D	DEC A
0DBB*	06 D0	0128D	RLCA
0DBD*	DD 21 000D*	0129D	LD C,A
00C1*	DD 09	0130D	LD B,0
00C3*	DD 66 DD	0131D	IX,MDVETB
0DC6*	DD 6E 01	D132D	LD IX,BC
00C9*	AF	0133D	ADD H,(IX)
D0CA*	77	0134D	LD L,(IX+1)
		0135D	XDR A
		D136D	LD (HL),A

Figure 8-2. Accessing a Simple Table

Here we didn't order the data in the table (namely, the five entries were not arranged in numerical sequence) but related them to an external index and accessed them by that index value.

Fixed Length Entry/ Variable Length Table

Another table that illustrates what we mean by "table driven" is shown in Figure 8-3. The GRIDTB of the figure is used in Chapter 15 to define a tic-tac-toe grid. There are 16 entries in the table, each one defining a line segment to be drawn by the DRAWL subroutine of that chapter. Each entry consists of five bytes and there are four fields within each entry.



**Figure 8-3. Fixed-Length Entry/
Variable-Length Table**

The first, second, and third fields of the entry are each one byte long, and the fourth field is two bytes long as shown in Figure 8-4. (The first field defines the graphics character to be stored; the second is 0 for a horizontal line or 1 for a vertical line; the third represents the number of character positions in the line; and the fourth field is the starting screen address for the line.)

+0	GRAPHICS CHARACTER
+1	0 = HORIZ 1 = VERT
+2	#CHARACTER POS'NS
+3	STARTING
+4	SCREEN ADDRESS

Figure 8-4. Entry Fields in GRIDTB

The last byte of the table is a **terminator** of -1. The terminator marks the end of the table and may be any value that is not a legitimate value for an entry in the table. You use the table by setting a pointer to the beginning and picking up the five bytes of each entry. After you have drawn the line, you increment the pointer by 5 and the code loops back to pick up the next entry. This continues until the -1 terminator is found. The code for accessing this table is shown in Figure 8-5.

```

0014' D0 21 0000* 00350 : DRAW GRID HERE
0018' D0 7E 00 00360 LO IX,GRITB ;TABLE FOR GRID
001B' FE FF 00 00370 ART005: LD A,(IX) ;GET CHARACTER
0010' 28 16 00380 CP OFFH ;TEST FOR TERMINATOR
001F' D0 4E 01 00390 JR Z,ART008 ;GO IF 00NE
0022' DD 46 02 00400 LD C,(IX+1) ;LOAO HORIZ/VERT
0025' 00 6E 03 00410 LO B,(IX+2) ;LOAO # OF CHAR POSNS
0028' DD 66 04 00420 LD L,(IX+3) ;START OF LINE, LSB
002B' C0 0000* 00430 LD H,(IX+4) ;START OF LINE, MSB
002E' 01 0005 00440 CALL DRAWL ;DRAW LINE
0031' D0 09 00450 LD BC,5 ;5 BYTES PER LINE
0033' 18 E3 00460 ADD IX,BC ;POINT TO NEXT LINE
00470 JR ART005 ;GO FOR NEXT LINE

```

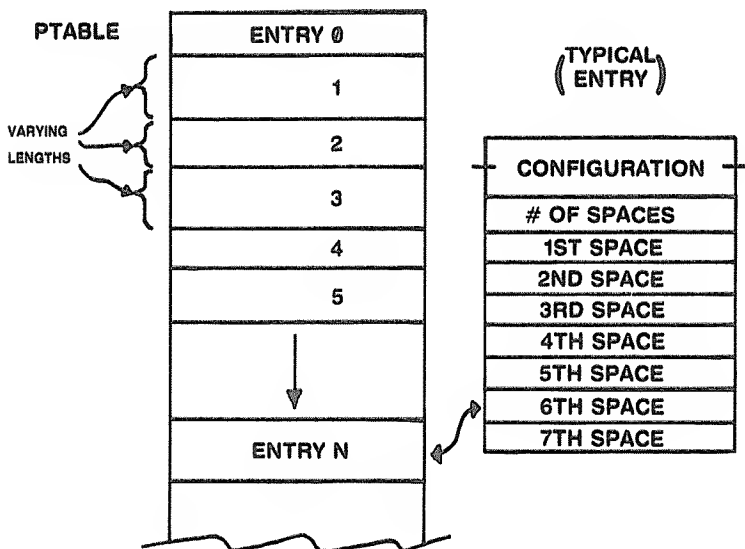
Figure 8-5. Table Use With Terminator

This somewhat “sloppy” table groups similar data together and helps “modularize” code. We could have incorporated all of the code to draw the grid into **in-line** code, but it’s a lot neater and efficient to put it into a table such as GRIDTB.

Variable Length Entry/ Variable Length Table

The tables above had **fixed-length entries**. In the case of MOVETB, the **number of entries** was fixed at five. GRIDTB had a variable number of entries, the end of the table being denoted by a terminator of -1.

An example of a table consisting of **variable-length entries** with a variable number of entries is shown in Figure 8-6. The PTABLE, or Permutation Table, is used in Chapter 15 to hold configurations of tic-tac-toe games.



**Figure 8-6. Variable-Length Entry/
Variable-Length Table**

The configuration for each tic-tac-toe game is held in the first two bytes of each entry in a special binary-coded form. The next byte defines the number of "spaces" in the tic-tac-toe configuration, from 3 to 9. The next 3 to 9 bytes represent a count related to each space. The length of each entry is therefore $2 + 1 + N$ where N is 3 to 9 or 6 to 12 bytes. When a search is made of the PTABLE, the length of the current entry must be computed by adding 3 plus the number found in the third byte.

PTABLE also has a variable number of entries, as the number of possible tic-tac-toe permutations is calculated **dynamically** (during program execution) by the tic-tac-toe program. In this case there is no terminator since a search of the PTABLE for a specific configuration must be successful. You don't have to have a terminator because the program will have a severe logic error if an entry is not found. The only action an unsuccessful search might produce would be a print-out message: FINISH DEBUGGING THE PROGRAM!!

Jump Tables

Another common type of table is a "jump table" or "branch table." This type of table is shown in Figure 8-7 in code from the MORG program of Chapter 14. It is used in similar fashion to a BASIC "computed GOTO" (ON N GOTO 100, 200, 300. . .). Its entries define the addresses of processing routines for the program where there are many different types of processing to be performed.

The FTAB is first **scanned** for the input code character. If it's found, the **index** to FTAB is then used to get the branch address from the BTAB. The code for the scan is shown in Figure 8-8.

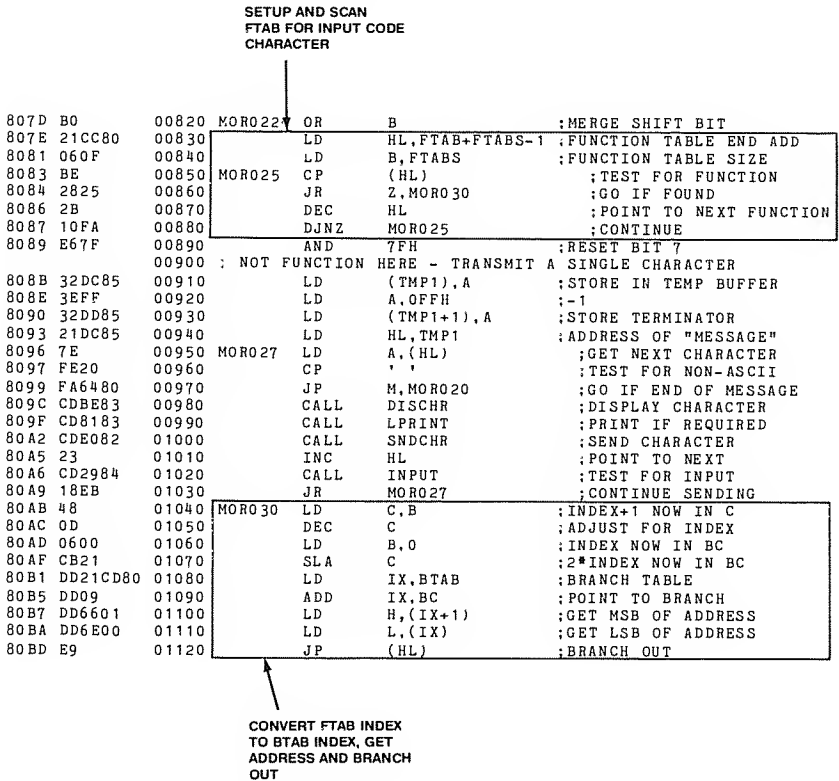


Figure 8-8. Jump Table Use

At MOR025, A holds the input character, and B holds the FTAB size, FTABS. FTABS is automatically computed at assembly time. The HL register holds a pointer to the **end** of the FTAB, set up by loading HL with FTAB+FTABS-1.

Hints and Kinks 8-1
Automatic Table Size

Programmers commonly use an expression like FTABS EQU \$-FTAB to have the assembler calculate table size. Entries can then be added in the table without having to change program constants. FTABS will be stored in the assembler symbol table with a value equal to the number of bytes in the table. As symbol table entries are 16 bits, this method works even for very long tables.

A CP is done to search the table. If the entry in FTAB does not compare, HL is decremented to point to the next lower entry. A DJNZ is then done back to MOR025. If the contents of B have been decremented down to zero, all entries have been compared and the search is unsuccessful. If the entry is found, the instruction at location MOR030 is executed.

At MOR030, the B register contains an **index** value to the FTAB character of 0 through 15 (FTABS). This index is transferred to BC, and then multiplied by two (SLA C). The IX register is now loaded with the start of BTAB, and BC is added to IX to point to the entry in BTAB corresponding to the FTAB entry. Note at this point, that IX only points to the BTAB entry; we have not picked up any address.

The next two instructions load H with the most significant byte of the entry and load L with the least significant byte of the entry. HL now contains the address of the processing routine from BTAB and a JP (HL) causes a jump out to the processing routine.

Using a branch table in this fashion is much cleaner than doing the equivalent **in-line** code of —

```
CP 'D'+80H ;TEST FOR DEFINE
JP Z,DEFINE ;GO IF DEFINE
CP 'S'+80H ;TEST FOR SPEED
```

JP Z,SPEED ;GO IF SPEED

,
,

— Hints and Kinks 8-2 — When to Use Branch Tables

Admittedly, there's a lot of ''overhead'' in branch table processing. One major difficulty is picking up a branch address. It would be nice to be able to use indexed addressing to pick up a 16-bit value instead of picking up the value one byte at a time.

Let's analyze the approach here. One way to branch out would have been

```
CP 'D'+80H ;IS THIS DEFINE MESSAGE
JP Z,DEFINE ;GO IF YES
CP 'S'+80H ;IS THIS DEFINE SPEED
JP Z,SPEED ;GO IF YES
```

This approach is clean and simple to debug. You would have used five bytes for each branch. The code shown in Figure 8-8 uses about 75 bytes, counting table storage. In this one example, then, 15 branch points would have been the ''break even'' point in terms of memory storage. Of course, there's also the factors of difficulty of coding and debugging, but it seems reasonable to use branch tables for anything over 20 or so entries. (Or 15 or so if you're writing a book on assembly language.).

Scanning

In the code above, we searched or scanned the FTAB for the key value. In this particular case, we scanned **backwards**. The usual procedure is to scan **forwards** through the table. The code of Figure 8-9 is a "Find Message" subroutine that searches a table of messages for

a given message number. The format of the Message Table MTAB is shown in Figure 8-10. It consists of ASCII characters, message numbers of 0 through 9, or -1 terminator values.

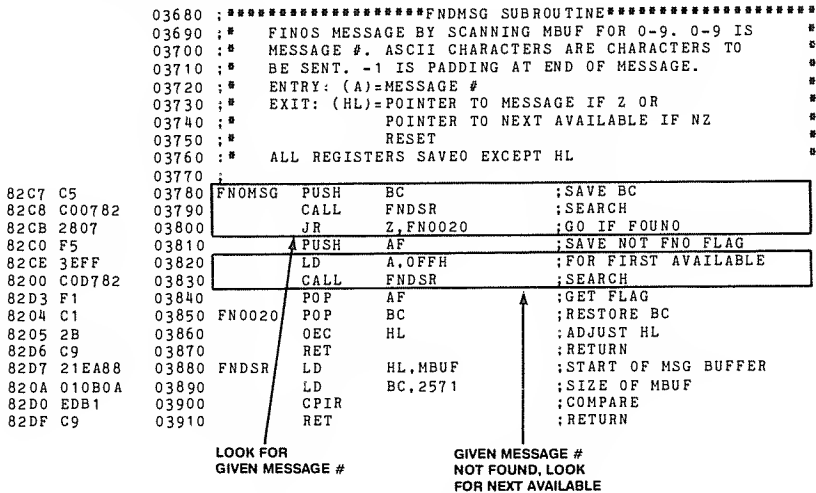
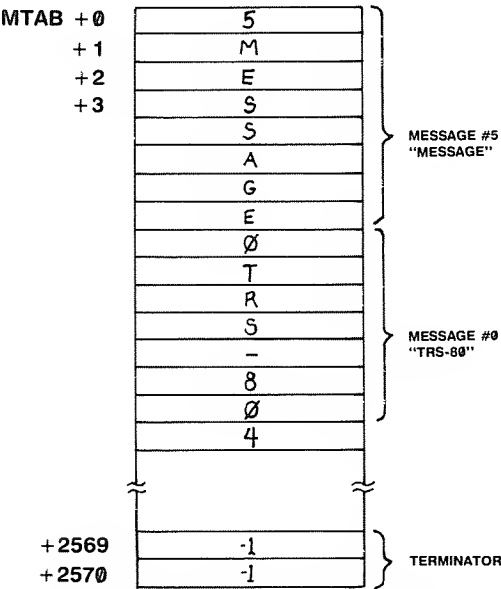


Figure 8-9. Scanning Example



You do the scan by a "block compare" instruction that compares the contents of A (the message #) with each byte of the table starting with MBUF and continuing through 2571 bytes. If the entry in MBUF compares, the Z flag is set, and HL points to the entry **plus one**.

— Hints and Kinks 8-3 —

Block Compares

The setup for a block compare looks like this:

- (HL) = Start of data for compare
- (BC) = Number of bytes in comparison area
- (A) = Search value

You then execute a CPIR instruction. It'll go through the block of memory for the number of bytes specified in BC and compare each memory byte with the search value. If a match is found, CPIR will stop with the Z flag set, and HL pointing one byte past the matching value.

(The reason for this is that the increment of HL is done before the comparison.) If the search value is not found in the memory block, the Z flag will not be set after the CPIR, and HL will point to the last byte plus one.

The CPDR performs much the same action but searches the block backwards. The CPI and CPD perform one comparison at a time, requiring a loop to be made back to the CPI or CPD.

The first part of the code looks for the given message number. If it isn't found (NZ), another CALL is made to FNDSR to search for the first -1 terminator. On exit, HL points to the message number if found or the first -1 byte. The Z flag is set if the message number was found.

The block compare instructions may be used conveniently to search a table of one-byte entries forwards or backwards (CPDR). They're very fast in comparison to other code sequences.

Ordered Tables

All of the above examples involved tables of **unordered** data. Scanning forwards or backwards through the table located one entry, an item at a time, until a match was found, or until the end (or beginning) of the table was reached. In the following discussion, we'll be concerned with operations on **ordered** tables.

Hints and Kinks 8-4

Ordered Tables

Tables are usually ordered in ascending order based on numeric 'weight.' One-byte entries pose no problem. Two-byte entry tables for numeric values will probably be in standard Z-80 16-bit address format, least significant byte followed by most significant byte. Greater than two-byte entries are not often found for numeric tables, but are common for strings.

ASCII strings in tables are also usually ordered on the basis of numeric weight. This means that you will use the hex equivalent of the ASCII value in determining whether one value is smaller than the next. This puts upper-case characters before lower case (TRS-80 before trs-80, for example and BIV,ROY G. after BIV ROY G).

The techniques for ordered tables (data entries are in **ascending** or **descending** order) are geared towards fast **searches** to find specific data items and **sorts** to organize the data in orderly fashion.

Searching

Suppose that we have a table of **ordered data**. We'll assume the table is made up of fixed-length entries, and it's a fixed number of entries. How do we search the entries in the table?

Sequential Search

The first way, of course, would be to scan the table **sequentially** as we have been doing. Even though assembly language is fast, the **search time** for a sequential search may become quite significant when large amounts of data are to be processed. We might be using the sequential search for a sort, for example. The entries in a table would be consecutively searched for the next smallest item and put into a second, sorted table. If there were 1000 entries, we'd have to search all 1000 to find the next smallest item, and we'd have to do that for 1000 items. This would mean 1000×1000 **iterations**, or about 1,000,000 iterations. If each iteration took 70 microseconds (about 12 instruction times), the total sort time would be $70 \text{ microseconds} \times 1,000,000$, or 70 seconds! Although this is the blink of an eye compared to the equivalent BASIC code, there is an occasional need for fast searches of ordered data. One of the "standard" high-speed search methods is the **binary search**.

Binary Search

In this search, the midpoint of the entries is compared to the **search key**. If the entry is greater than the search key, the top half of the table is discarded and the next comparison is made in the midpoint of the bottom half; if the entry is less than the sort key, the bottom half of the table is discarded and the next comparison is made in the midpoint of the top half. This division by 2 continues for smaller and smaller segments of the table until the entry is found or until the last segment (one entry!) has been compared and no match has been made (see Figure 8-11).

**SORTED
TABLE**

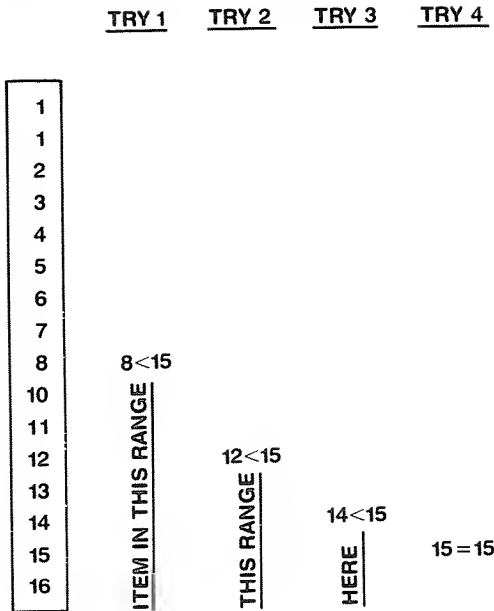


Figure 8-11. Binary Search Algorithm

Figure 8-12 shows this binary search technique implemented in an assembly-language program. BINSRC is designed to search a table made up of 16-bit entries for a given search value. The entries are ordered in ascending fashion and are in standard Z-80 16-bit format, least significant byte followed by most significant byte.

```

8000      00100      ORG      8000H
00110      ;*****
00120      ;*      BINARY SEARCH OF 16-BIT ENTRY TABLE      *
00130      ;* ENTRY: (OE)=# OF ENTRIES                        *
00140      ;*      (BC)=SEARCH VALUE                          *
00150      ;*      (HL)=START OF TABLE                      *
00160      ;* EXIT: (HL)=ENTRY LOCATION OR -1 IF NOT FOUND    *
00170      ;*      ALL OTHER REGISTERS SAVE                  *
00180      ;*****
8000 F5      00190      BINSRC      PUSH      AF      ;SAVE REGISTERS
8001 05      00200      PUSH      OE
8002 00E5    00210      PUSH      IX
8004 226680 00220      LO      (START),HL      ;SAVE TABLE START
8007 E0535E80 00230      LO      (HI),OE      ;HIGH VALUE
800B 110000   00240      LO      OE,0        ;ZERO
800E E0536080 00250      LO      (LO),OE      ;LOW VALUE

```

```

8012 E05B6080 00260 BIN010 LO OE,(LO) ;GET LOW VALUE
8016 2A5E80 00270 LO HL,(HI) ;GET HIGH VALUE
8019 B7 00280 OR A ;CLEAR CARRY
801A E052 00290 SBC HL,OE ;FINO HIGH-LOW
801C CB3C 00300 SRL H ;MSB/2
801E CB10 00310 RR L ;(HI-LOW)/2
8020 226480 00320 LO (INCM),HL ;SAVE INCREMENT/2
8023 19 00330 AOO HL,OE ;LO+1/2INC
8024 226280 00340 LO (MIO),HL ;SAVE MIOPOINT
8027 29 00350 AOO HL,HL ;MIO*2 FOR WORO ADDRESS
8028 E05B6680 00360 LO OE,(START) ;START OF TABLE
802C 19 00370 AOO HL,OE ;START + MIOPOINT AOO
8020 E5 00380 PUSH HL ;PUT IN IX
802E 00E1 00390 POP IX
8030 D06601 00400 LO H,(IX+1) ;GET MSB BYTE
8033 006E00 00410 LO L,(IX) ;GET LS BYTE
8036 B7 00420 GR A ;CLEAR CARRY
8037 E042 00430 SBC HL,BC ;TEST VALUE
8039 2A6280 00440 LO HL,(MIO) ;SETUP FOR STORE
803C 2813 00450 JR Z,BIN030 ;GO IF FOUNO
803E 300C 00460 JR NC,BIN020 ;GO IF LOW
8040 226080 00470 LO (LO),HL ;NEW LOW
8043 2A6480 00480 BIN015 LO HL,(INCM) ;GET INCREMENT
8046 7C 00490 LO A,H ;O FOR SMALLEST TEST
8047 B5 00500 OR L ;CLEAR CARRY
8048 280F 00520 JR Z,BIN040 ;GO IF ENO
804A 18C6 00530 JR BIN010 ;LOOP
804C 225E80 00540 BIN020 LO (HI),HL ;NEW HIGH
804F 18F2 00550 JR BIN015 ;LOOP
8051 00E5 00560 BIN030 PUSH IX ;ADDRESS
8053 E1 00570 POP HL ;FOR RETURN
8054 00E1 00580 BIN035 POP IX ;RESTORE REGISTERS
8056 01 00590 POP OE
8057 F1 00600 POP AF
8058 C9 00610 RET ;RETURN
8059 21FFFF 00620 BIN040 LO HL,-1 ;OUMMY FOR NOT FOUNO
805C 18F6 00630 JR BIN035 ;GO TO RETURN
0002 00640 HI OEFS 2
0002 00650 LO OEFS 2
0002 00660 MIO OEFS 2
0002 00670 INCM OEFS 2
0002 00680 START OEFS 2
0000 00690 ENO
00000 TOTAL ERRORS

```

Figure 8-12. Binary Search Routine

The routine is entered with DE holding the number of **entries** in the table, BC holding the 16-bit search value, and HL pointing to the table start. A binary search will be made of the table. If the value in BC is found, HL is returned with a pointer to the table entry. If there is no corresponding table entry, HL is returned with a "not found" value of -1.

Data in this table must be 16-bit **unsigned** values. All compares will be of unsigned data (8000H is greater than 7FFFH and FFFFH is greater than FFFEh, for example).

BINSRC works as follows. Five variables are used — HI, LO, MID, INCM, and START. START is simply the table start from entry. HI holds the current high index value defining the top of the range. LO holds the low index value. MID holds $(HI-LO)/2 + LO$, which points to the next “test” value in the middle of the range. INCM is $(HI-LO)/2$. INCM gets smaller and smaller as the search “zeroes in” on a value.

The routine starts with HI equal to the number of entries and LO equal to 0. For each iteration, HI-LO is found, divided by two, and added to LO. This gives the **index value** for the comparison. This value is multiplied by two and added to START to find the actual table address.

After the address is found, the table entry is compared to the BC search value. If the table value is less, the MIDpoint value replaces LO; if the table value is more, the MIDpoint value replaces HI. Either way, one half of the current range is discarded.

The “halving” process continues until the search value is found (BIN030) or until the increment INCM is zero, indicating that $HI-LO=1$. (The test is made after the comparison.)

With the proper parameter passing setup, BINSRC could be used to search a BASIC integer array, as the array would be made up of 16-bit values. Use VARPTR to find the array address.

There are other types of searches, but the binary search and sequential searches are the most commonly used in assembly language. Many searches use the sequential scan since it's the easiest to implement.

Sorting

Now the question arises, "How did the data in a table get sorted in the first place?" There are a number of ways you can sort data — the brute-force two-buffer sort, bubble sorts, binary-insertion sorts, the Shell-Metzner sort, and many others. We'll discuss the first two, which should handle most assembly-language applications.

The Brute-Force Two-Buffer Sort

In this sort we require two buffers. The second is the same size as the first. Here you scan the first "buffer," which is the table of items to be sorted, sequentially for the smallest data item. When it's found, it's put into the second buffer in the next position. The item is then "blanked out" in the first buffer.

This sort requires a lot of memory space because of the two buffers and is relatively slow because of the complete scan required for every item. On the other hand, it's simple in concept and easy to debug.

The assembly-language subroutine for the sort is shown in Figure 8-13. Here again, the sort works with 16-bit data values. Why 16 bits? The 8-bit case is virtually useless, and a version that handles long strings is much more complicated. The 16-bit version is complicated enough due to 16-bit comparisons that must be done and other unwieldy 16-bit operations.


```

8DD0      DD1DD      ORG      800DH
00110     ;*****
0D120     ;      TWD BUFFER SDRT OF 16- IT ENTRIES
00130     ; ENTRY: (IX)=BUFFER 1 ADDRESS
DD14D     ;      (IY)=BUFFER 2 ADDRESS
0D150     ;      (DE)=NUMBER OF ENTRIES
0016D     ; EXIT: BUFFER1 ENTRIES SORTED IN BUFFER2. BUFFER1
0017D     ;      DESTROYED
00180     ; NOTE: ENTRIES OF 0 ARE NOT ALLDWD
00190     ;      ALL REGISTERS *DESTROYED*
00200     ;*****
8000 DD225B8D 00210 TWDBUF LD (BUF1),IX ;SAVE START
8DD4 ED535D80 0022D LD (COUNT),DE ;SAVE COUNT
8008 DD2A5B80 0023D LD IX,(BUF1) ;LOAD START
80DC 210D0D 0024D LD HL,D ;ZERO
8DDF 22618D 0025D LD (CURLOC),HL ;CURRENT LDCATIDN
8D12 21FFFF DD26D LD HL,OFFFHH ;FOR SMALLEST
8015 225F80 0027D LD (CURVAL),HL ;CURRENT VALUE
8D18 ED5B5D80 0028D LD DE,(CDUNT) ;GET COUNT
8D1C DD6601 0029D TWBD1D LD H,(IX+1) ;GET MSB
801F DD6EDD 0030D LD L,(IX) ;GET LSB
8022 7C 0031D LD A,H ;TEST FOR D ENTRY
8023 B5 0032D OR L
8D24 2811 0033D JR Z,TWBD30 ;GD IF ZERO
8D26 ED4B5F8D 0034D LD BC,(CURVAL) ;GET CURRENT VALUE
802A B7 0035D DR A ;CLEAR CARRY
802B ED42 DD36D SBC HL,BC ;COMPARE
802D 3008 0037D JR NC,TWB030 ;GO IF CURRENT LARGER
802F 09 0038D ADD HL,BC ;RESTORE NEW VALUE
8D3D 225F8D DD39D LD (CURVAL),HL ;NEW SMALLEST
8033 DD226180 0040D LD (CURLOC),IX ;NEW LDCATIDN
8037 DD23 0041D TWBD3D INC IX ;POINT TD NEXT
8039 DD23 0042D INC IX
8038 1B 0043D DEC DE ;DECREMENT COUNT
803C 7A 0044D LD A,D ;TEST FOR ZERO
8D3D B3 0045D DR E
803E 20DC 0046D JR NZ,TWB010 ;GO IF NOT END
8040 2A6180 0047D LD HL,(CURLOC) ;GET CURRENT LOCATION
8043 7C DD48D LD A,H ;TEST FOR 0
8D44 B5 0049D OR L
8D45 C8 DD5DD RET ;RETURN IF ALL BLANKED
8D46 AF 0050D XOR A ;ZERO TD A
8D47 3E00 0051D LD A,D ;0 TO A. DDN'T SET CARRY
8049 77 DD52D LD (HL),A ;BLANK ENTRY
804A 23 0053D INC HL
804B 77 0054D LD (HL),A
8D4C 2A5F80 DD55D LD HL,(CURVAL) ;GET CURRENT VALUE
8D4F FD7401 DD56D LD (IY+1),H ;STORE MSB
8052 FD750D 0057D LD (IY),L ;STORE LSB
8055 FD23 DD58D INC IY ;BUMP BUFFER2 PNTR
8057 FD23 0059D INC IY
8059 18AD 0060D JR TWB005 ;CDNTINUE
0002 0061D BUF1 DEFS 2
0002 0062D COUNT DEFS 2
0D02 0063D CURVAL DEFS 2
0002 0064D CURLOC DEFS 2
0000 0065D END
00D00 TDTAL ERRORS

```

**Figure 8-13. Two-Buffer
Sort Routine**

The routine is entered with IX and IY pointing to the two buffers. IX points to the buffer containing the unsorted data; IY points to a second buffer holding the sorted data. As the entries in the IX buffer are moved to the IY buffer,

they are "blanked" by filling with a zero value. For this reason, zero is not a valid value in the IX buffer, unless you'd like it to be ignored.

DE contains the number of entries in each of the buffers. The routine first scans the IX buffer until it finds the smallest entry. A dummy count of 0FFFFH is initially put in variable CURVAL to guarantee that a smaller entry will be found. Each entry in the IX buffer is compared with the smallest entry. At the end of the scan, the smallest entry is in CURVAL and its address is in CURLOC. The entry is now blanked by filling its position with 0000H. The entry is then stored in the next position in the IY buffer. The process is repeated until all entries in the IX buffer have been blanked. Figure 8-14 shows the sort action.

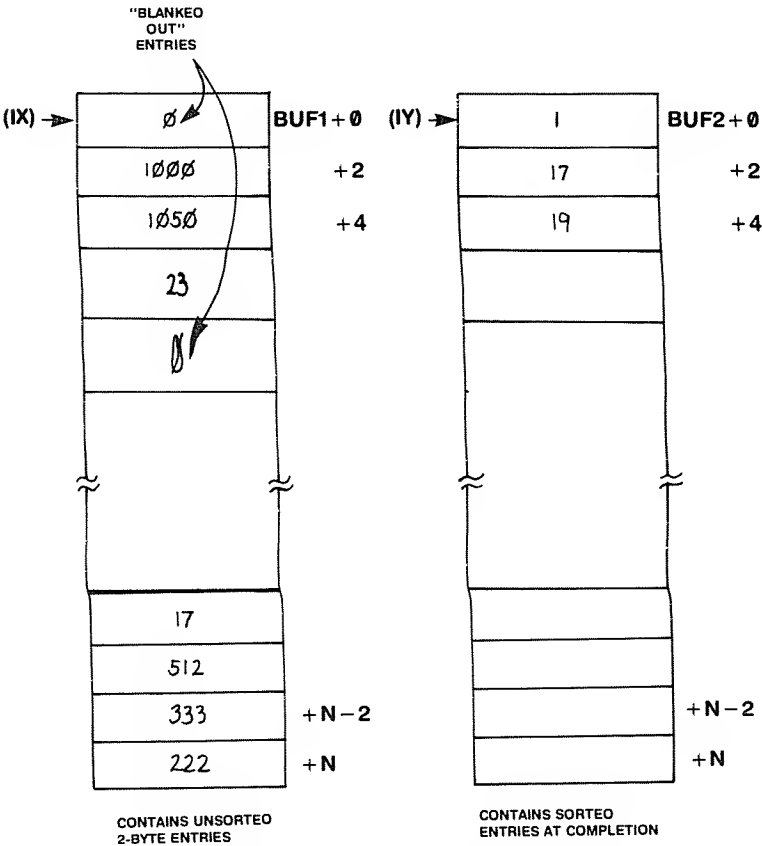


Figure 8-14. Two-Buffer Sort Action

Hints and Kinks 8-5

Two-Buffer Sort Speed and Storage

The two-buffer sort is very expensive in terms of memory storage since it must use another buffer the same size as the first to do its dirty work. How fast is it? Because the sort must pass completely through the first buffer for each entry, it requires n passes to sort, where n is the number of entries. Each iteration requires approximately 35 instructions. If we estimate about 5 microseconds per instruction, that means each pass takes about 175 microseconds. For a thousand entry table, this is a little under a fifth of a second. Quite a difference from BASIC - we can afford to be 'sloppy' in high-speed code such as this. Nevertheless, many sorts are run 'off-line' in commercial programming departments during hours when only the maintenance crew is around.

The Bubble Sort

The "bubble" sort is another sorting technique. It uses only one buffer and is therefore efficient in memory storage. The bubble sort works with two entries of the table at a time. It switches the entries if the second entry is of lower value than the first. In this way, the "lighter" entries bubble to the top. After each pass through the table, another pass is made. If no switches have occurred after any pass, the sort is done (See Figure 8-15).

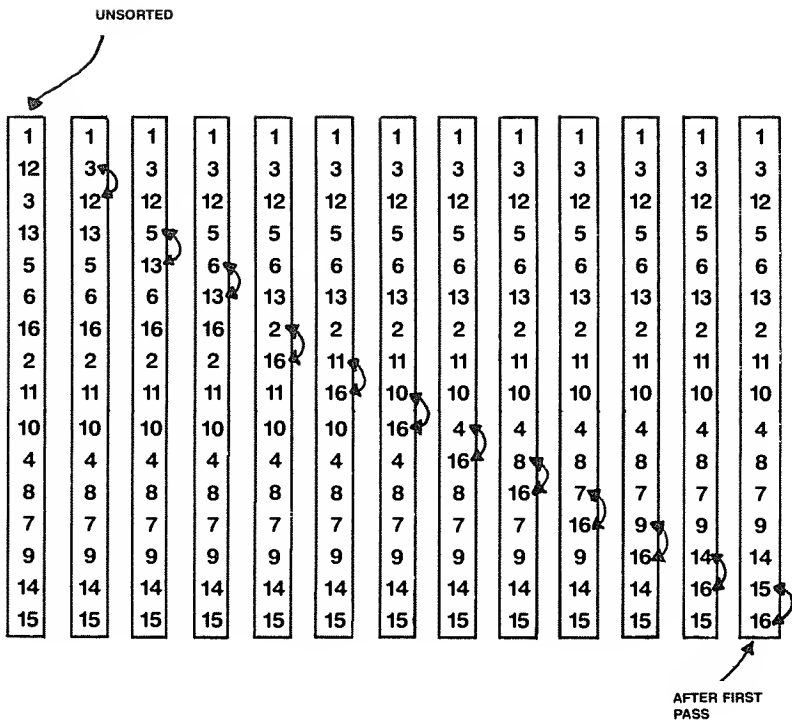


Figure 8-15. Bubble Sort Action

The assembly-language code for the bubble sort is shown in Figure 8-16. The bubble sort is an elegant, clean code but is somewhat slow for all its grace. Pass after pass is made through the buffer, and each adjacent set of entries is compared. If the “top” element is larger than the “bottom,” the two are swapped. A change flag (CHNG) is set if any swap is made. If no swap has been made in any pass, the swapping is over, and the sort is done. The BUBBLE routine here is a good example of the power of indexing.

The BUBBLE routine is a great one to run with the contents of the buffer equated to video display memory if you'd like to see a graphic example of the sorting process. Set IX equal to 3C00H and DE equal to 200H (there are two bytes per entry). You'll have to call the routine from another bit of assembly-language code, or "dummy up" a CALL from DEBUG, such as CD 00 80 C3 03 A0 (CALL 8000H and Endless Loop at A003H).

```

8000          00100          ORG      8000H
00110          ;*****
00120          ;          BUBBLE SORT
00130          ; ENTRY: (IX)=BUFFER
00140          ;          (DE)=NUMBER OF ENTRIES IN 16-BIT ENTRY TABLE
00150          ; EXIT:  BUFFER CONTAINS SORTED ENTRIES
00160          ; ALL REGISTERS *DESTROYED*
00170          ;*****
8000 DD224A80 00180 BUBBLE LD      (BUFFER),IX      ;SAVE START
8004 1B       00190 DEC      DE                      ;DECREMENT COUNT
8005 ED534C80 00200 LD      (COUNT),DE          ;SAVE FOR PASSES
8009 AF       00210 BUB010 XOR      A              ;"CHANGE" FLAG
800A 324E80   00220 LD      (CHNG),A              ;TO ZERO
800D ED4B4C80 00230 LD      BC,(COUNT)          ;LOAD #-1
8011 DD2A4A80 00240 LD      IX,(BUFFER)          ;GET START OF BUFFER
8015 DD6601   00250 BUB015 LD      H,(IX+1)        ;GET MSB OF FIRST
8018 DD6E00   00260 LD      L,(IX)              ;GET LSB OF FIRST
801B DD5603   00270 LD      D,(IX+3)            ;GET MSB OF SECOND
801E DD5E02   00280 LD      E,(IX+2)            ;GET LSB OF SECOND
8021 B7       00290 OR      A                      ;RESET CARRY
8022 ED52     00300 SBC      HL,DE              ;COMPARE BY SUB
8024 2814     00310 JR      Z,BUB020            ;DON'T SWAP IF EQUAL
8026 3812     00320 JR      C,BUB020            ;GO IF FIRST SMALLER
8028 19       00330 ADD      HL,DE              ;RESTORE HL
8029 DD7201   00340 LD      (IX+1),D            ;SWAP FIRST WITH SECOND
802C DD7300   00350 LD      (IX),E
802F DD7403   00360 LD      (IX+3),H
8032 DD7502   00370 LD      (IX+2),L
8035 3E01     00380 LD      A,1                ;NON-ZERO
8037 324E80   00390 LD      (CHNG),A            ;SET CHANGE FLAG
803A DD23     00400 BUB020 INC      IX            ;BUMP PNTR
803C DD23     00410 INC      IX
803E 0B       00420 DEC      BC                ;DECREMENT COUNT
803F 78       00430 LD      A,B                ;TEST COUNT
8040 B1       00440 OR      C
8041 20D2     00450 JR      NZ,BUB015          ;GO IF NOT LAST
8043 3A4E80   00460 LD      A,(CHNG)          ;GET FLAG
8046 B7       00470 OR      A                  ;TEST CHANGE
8047 C8       00480 RET      Z                ;RETURN IF NONE
8048 18BF     00490 JR      BUB010            ;AT LEAST ONE CHANGE
0002          00500 BUFFER DEFS      2
0002          00510 COUNT DEFS      2
0001          00520 CHNG  DEFS      1
0000          00530 END
00000 TOTAL ERRORS

```

Figure 8-16. Bubble Sort Routine

Hints and Kinks 8-6
Bubble Sort Speed and Storage

The bubble sort is very efficient in terms of memory storage since it works within the given buffer and uses very little storage elsewhere. It's a good sort to use when space is at a premium.

However, speed can be another matter. There is no fixed number of iterations through the table with a bubble sort. The best case would be one pass, when all entries are already sorted. The worst case (I believe) is the case where the entries are completely reversed with the 'lighter' elements at the bottom of the table, water-sogged. If even one element in the table must be moved from bottom to top, it will take $n-1$ iterations, where n is the size of the table in entries. A rough estimate of the time for our bubble sort would be 25 instructions per iteration at 5 microseconds per instruction, or 125 microseconds per iteration. Moving one element from bottom to top in a 1000-entry table would require about an eighth of a second. (Still not bad compared to BASIC)

There are a number of other sorts that could be used in assembly language, but they're somewhat more complex to code and should be used only if you'd really like to crank out every last bit of speed in your assembly-language sorting. Hopefully these sorts have whetted your interest.

Chapter Nine

Graphics Display Processing

Several chapters ago we looked at some of the techniques used to display character data on the screen. Here, we'll talk about graphics display processing — how to put graphics characters on the screen, how to draw shapes and lines, and how to “animate” screen images.

Graphics Characteristics Character Data Storage

Most of you are familiar with the scheme used in the TRS-80 for displaying graphics, but we'll go over it again before we talk about graphics processing. There're 1024 **character positions** on the screen, arranged in 16 lines of 64 characters each. In the “upper case” version of the TRS-80, video memory stores only seven bits of data for each character stored. As a matter of fact, all character data must have the most significant bit, (bit 7), set to 0, so the video memory only uses six bits to represent the ASCII character codes. The “missing bit” is bit 6. Figure 9-1 shows how the 64 valid ASCII codes are stored in memory.

<u>CHARACTER</u>	<u>ASCII</u>	<u>MEMORY*</u>	<u>CHARACTER</u>	<u>ASCII</u>	<u>MEMORY*</u>
SP	20H	20H	@	40H	00H
!	21	21	A	41	01
"	22	22	B	42	02
#	23	23	C	43	03
\$	24	24	D	44	04
%	25	25	E	45	05
&	26	26	F	46	06
'	27	27	G	47	07
(28	28	H	48	08
)	29	29	I	49	09
*	2A	2A	J	4A	0A
+	2B	2B	K	4B	0B
,	2C	2C	L	4C	0C
-	2D	2D	M	4D	0D
.	2E	2E	N	4E	0E
/	2F	2F	O	4F	0F
0	30	30	P	50	10
1	31	31	Q	51	11
2	32	32	R	52	12
3	33	33	S	53	13
4	34	34	T	54	14
5	35	35	U	55	15
6	36	36	V	56	16
7	37	37	W	57	17
8	38	38	X	58	18
9	39	39	Y	59	19
:	3A	3A	Z	5A	1A
;	3B	3B	↑	5B	1B
<	3C	3C	↓	5C	1C
=	3D	3D	←	5D	1D
>	3E	3E	→	5E	1E
?	3F	3F	—	5F	1F

*MEMORY IS UPPER CASE WITHOUT LOWER-CASE OPTION

Figure 9-1. Video Memory Codes

When the video memory byte holding character data is read, the memory form of the data is converted back to ASCII by setting bit 6 if bit 5 AND bit 7 are 0s. Since bit 7 is never set for character data, the video-memory codes are reconverted back to ASCII as shown in Figure 9-2.

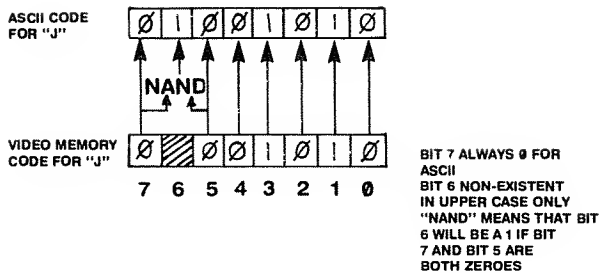


Figure 9-2. Reconversion From Video Memory

When character data from 00000000 through 00011111 (the control codes) or from 01100000 through 01111111 (lower case) are stored in video memory, they lose bit 6 and become the characters shown in Figure 9-1. Attempting to store codes 0 through 127 results in reading back 2 sets of the 64 valid character codes. The moral to this long tale is "Never store anything in upper case video memory except ASCII data (or graphics data, as we'll see) unless you want it to be converted to ASCII!"

Hints and Kinks 9-1 Lower Case Modification

The lower case modification adds another RAM chip to the video memory and a new character generator chip. The RAM chip adds an eighth bit and eliminates the special video memory storage codes; all data is then stored in video memory in 8-bit format. The NAND logic for bit 6 is also deleted.

Storage of Graphics Data

Graphics data is stored exactly the same as character data. Bit 6 is lost in upper case versions of the TRS-80. However, here bit 7 is set to indicate that the data does not represent 64 ASCII-like codes, but 64 graphics configurations. The graphics configurations allowed are shown in Figure 9-3, along with their data values.

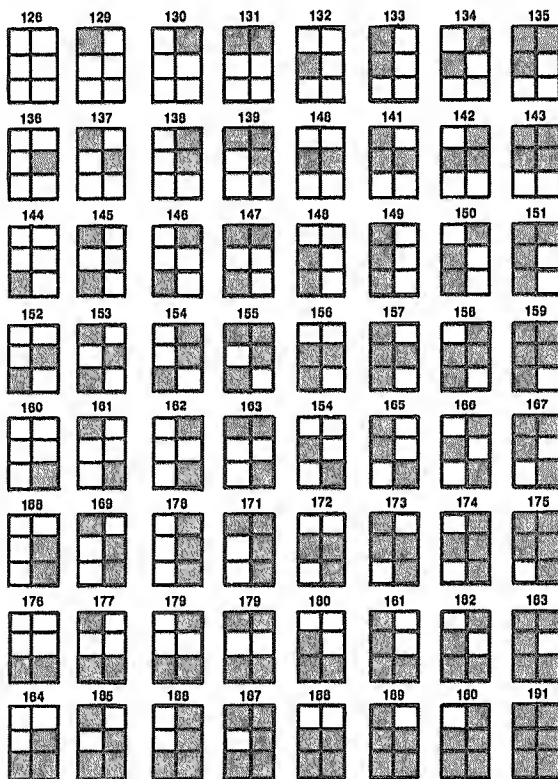


Figure 9-3. Graphics Codes

When the data is read back from video memory for refresh (hardware display of memory on the screen), the CPU hardware logic examines bit 7 to see if it's a 0 or 1. If it's a zero, the 6 bits are input to a graphics character generator chip that converts the 6 bits to a 5 by 7 gra-

phics character. If it's a one, the character generation is disabled, and each of the six bits produces one **pixel** worth of data, on or off, as shown in Figure 9-4.

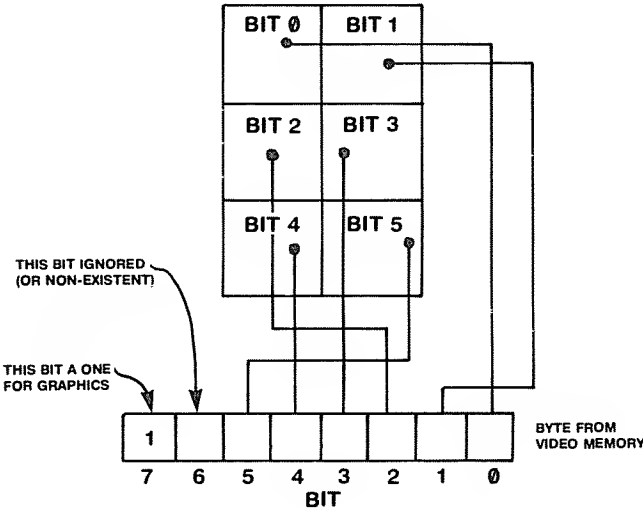
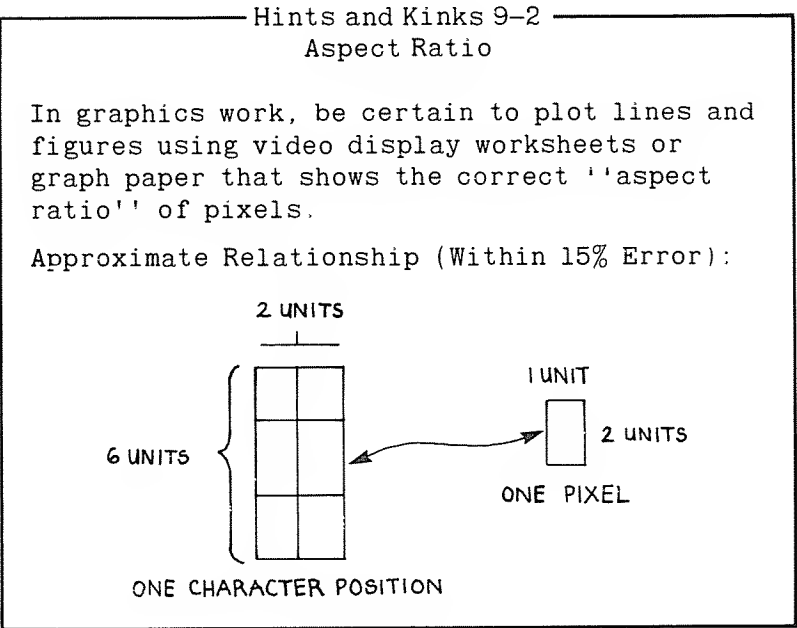


Figure 9-4. Pixel Representation

As each character position on the screen is divided into six pixels, there are $1024 \div 6 = 6144$ pixel positions on the screen, divided into 128 columns and 48 rows, as shown in Figure 9-5.

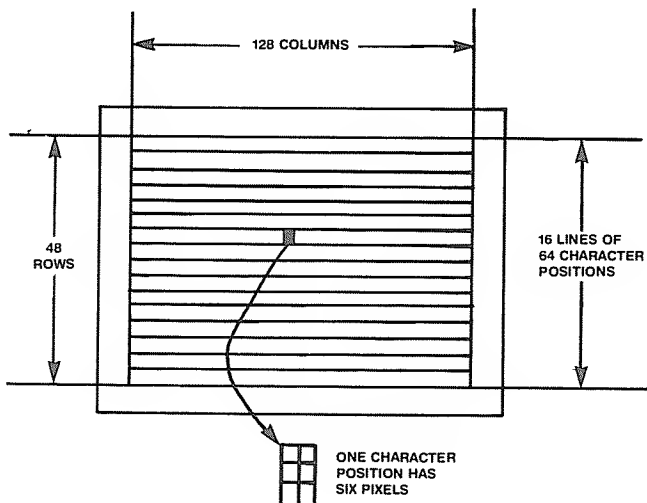


Figure 9-5. Graphics and Character Positions

Character Position and Pixel Addressing

We saw in Chapter 7 that it's very easy to output character data to the screen. The ASCII character you want to display is simply stored in the proper character position. Compute the character-position address by adding the character position number 0 through 1023 to the address of the start of video memory, 3C00H.

Even when you plan to store a character at a specific line and character position within a line, the address computation is simple. The formula for a video-memory address for a line character-position address is:

$$\text{ADDRESS} = \text{LINE} \times 64 + \text{CHAR POS} + 3C00H$$

where LINE is the line number 0 through 15 and CHAR POS is the character position within the line 0 through 63.

Addressing pixels, however, isn't that easy. As a matter of

fact, it's a chore due to the **memory mapping** of the video memory. As our seven-year old TRS-80 afficianado puts it, "Six into eight don't go!"

Random addressing of pixels requires an x,y address, where $x = 0$ through 127 and $y = 0$ through 47. This is familiar to most readers as the BASIC SET/RESET format. Each pixel is in one of the 1024 bytes of video memory; within the byte, the pixel may be in **bit positions** 0 through 5. The task in addressing a pixel is to compute the **byte address** of the byte containing the pixel, followed by the **bit address** of the pixel within the byte.

If we do some thinking about this problem, we can see that the byte address is given by:

$$\text{BYTE ADDRESS} = 3C00H + (Y/3)Q*64 + (X/2)Q$$

This formula says if we take the Y address and divide by three, the quotient (Q) will represent the **line number** (0 through 15) containing the pixel. Multiplying this line number by 64 will give the **displacement** from the start of video memory of the line. If the X address is divided by two, the quotient will give the **character position** containing the pixel along the line. The actual byte address is now $3C00H + \text{line displacement} + \text{character displacement}$.

We now have the byte address, but what about the **bit address** within the byte? The row address is given by:

$$\text{BIT ADDRESS} = (Y/3)R*2 + (X/2)R$$

The remainder (R) of $Y/3$ gives the **row number** of the bit. Since there are two bits per row, this must be multiplied by two to give the row displacement. The remainder of $X/2$ gives the **column number** of the bit. Adding this to the row displacement gives the bit position 0 through 6 of the bit.

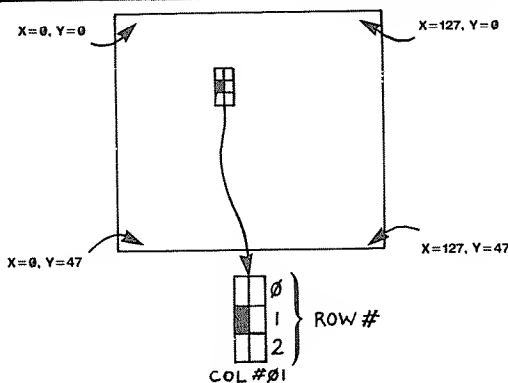
The formulas above are not easy to visualize, but if you draw a sketch of the screen showing character positions and pixel positions and try some examples, you'll see how they were derived. Figure 9-6 shows a recap of pixel addressing.

Hints and Kinks 9-3 Higher Speed SET/RESET

Is a higher-speed SET/RESET possible? A brute force method might employ a table look up indexed by $Y*128+X$. Each entry in the table would be a byte address of 0-1023 and a bit address of 0-7. This could be stored in two bytes, and the entire table would require $6144*2=12288$ bytes.

You could modify this table scheme on a line basis by dividing Y by 16 (easy to do) and finding $(Y/16)R*128+X$. This index would access a 384-entry table of two bytes each (1068 bytes) containing the byte address and bit address. The actual byte address could be found by: $TABLE\ BYTE\ ADDRESS + (Y/3)Q*64 + 3C00H$.

Estimated speed for this SET/RESET would be somewhere around 7500 points per second.



1. $LINE \# = \frac{Y}{3}$ QUOTIENT (EACH LINE OF 16 HAS 3 ROWS OF Xs)
2. $ROW \# \text{ OF CHARACTER} = \frac{Y}{3}$ REMAINDER (ROW # WITHIN THE CHARACTER)
3. $CHARACTER \text{ POSITION} = \frac{X}{2}$ QUOTIENT (FROM START OF LINE)
4. $COLUMN \# \text{ OF PIXEL} = \frac{X}{2}$ REMAINDER (0 OR 1)
5. $BYTE \text{ DISPLACEMENT FROM START OF VIDEO MEMORY} = (LINE \#) * 64 + CHARACTER \text{ POSITION}$
6. $ACTUAL \text{ LOCATION} = (LINE \#) * 64 + CHARACTER \text{ POSITION} * 3C00H$
7. $BIT \text{ POSITION WITHIN CHARACTER} = (ROW \#) * 2 + COLUMN \text{ NUMBER}$

Figure 9-6. Pixel Addressing

Random Vs. Character Position Graphics

There are two basic methods of writing out graphics data to the screen: "character position" graphics and "random" graphics.

In random graphics, a point is addressed by its x,y coordinates and either set, reset, or tested. This method of displaying graphics data requires a **graphics driver** program to convert the x,y coordinates into a byte and bit address as we discussed above. Random graphics are used for plotting, animation, line drawing, and the like.

There's a lot of graphics display work, however, that can be done by character-position-oriented graphics. This still involves setting pixels, but the pixels are not addressed randomly. The pixel positions to be set (or reset) are known beforehand, and data is output on a character position basis to produce the graphics.

We'll discuss these methods before getting into the more complicated random techniques.

Character-Oriented Graphics

Horizontal and Vertical Line Drawing

The simplest graphics processing involves drawing a horizontal or vertical line. Since there are three rows in a graphics character position, the horizontal line height can be either 1/3, 2/3, or 3/3 of a character position height. The width of a vertical line can be 1/2 or 2/2 of a character position width.

Figure 9-7 shows some code from the MORG program of Chapter 13 that draws a thick line across line 12 of the screen. `LINE12` is equated to `3F00H`. The `FILLCH` (Fill Character) subroutine is used to fill an `8FH` for 64 bytes, starting at `3F00H`.

8026	3E8F	00470	LD	A,0BFH	:ALL ON GRAPHICS CHAR
8028	11003F	00480	LD	DE,LINE12	:LINE 12
802B	014000	00490	LD	BC,64	:# OF BYTES
802E	CD8385	00500	CALL	FILLCH	:DRAW LINE

**Figure 9-7. Horizontal Line
Drawing**

When you need to draw a vertical line, the process is similar, except that the "increment" is 64 instead of one. This sets the next character position under the last to draw the line vertically. Code for drawing a vertical line is shown in Figure 9-8 from the DRAWL subroutine of Chapter 14. The graphics character to be used is in the A register.

000F'	F1	00350	:VERTICAL LINE HERE		
0010'	11 0040	00360	DRA060:	POP AF	:RESTORE CHARACTER
0013'	77	00370	LD	DE,64	:INCREMENT
0014'	19	00380	DRA065:	LD (HL),A	:STORE GRAPHICS
0015'	10 FC	00390	ADD	HL,DE	:POINT TO NEXT
		00400	DJNZ	DRA065	:GO IF MORE

**Figure 9-8. Vertical Line
Drawing**

If you're doing a large amount of line drawing in a program, then it'd be convenient to automate the process somewhat. One approach is to make a table of lines to be drawn. This makes it easy to define new displays and to correct existing ones (another example of a "table-driven" approach).

Hints and Kinks 9-4
Graphics Driver

A table-driven "line drawer" could be expanded even further. A graphics driver subroutine could decode such table entries as 'draw horizontal line,' 'draw vertical line,' 'draw rectangle with given corners,' 'draw filled in box,' 'draw diagonal line,' and so forth. This would be a useful program if you continually do display work, but probably not worth the time otherwise.

We used this method in Chapter 14 for the Tic-Tac-Toe program. The programmer used a "grid table," GRIDT, to define all the lines to be drawn in the Tic-Tac-Toe grid. (There were more than four lines, due to some segments that occupied less than a character position.)

Each entry in the GRIDTB is five bytes long (see Figure 8-4) and defines the graphics character to be used (one byte); a code for horizontal/vertical (one byte); the number of character positions to be used (one byte); and the line starting position (two bytes). The table is terminated by a minus one. The DRAWL subroutine shown in Figure 9-9 is called for each entry in the table to draw a single line, horizontal or vertical.

Although the routines here involved only **setting** pixels, it's easy to see how codes for resetting pixels could also be incorporated into code or table-driven routines.

Resetting the pixels could be done either by clearing the screen with graphics 80H characters (about 1/25th second) and redrawing a new line or by going back over the old line with a "fill" byte of 80H. The routines above allow horizontal or vertical lines to be drawn at rates of about 3000 lines per second (assuming average line lengths of 20 character positions) which would permit fast game displays or other display processing.

```

00100      TITLE      DRAWSR
00110      ENTRY      DRAWL
00120      ;*****
00130      ;***** ORAW LINE SUBROUTINE *****
00140      ;*****
00150      ;***** ORAWS HORIZONTAL OR VERTICAL LINE *****
00160      ;***** ENTRY: (A)=GRAPHICS CHARACTER TO BE USED *****
00170      ;***** (C)=0 IF HORIZONTAL, NON-ZERO IF VERTICAL *****
00180      ;***** (HL)=SCREEN START POSITION *****
00190      ;***** (B)=# OF CHARACTER POSITIONS TO BE USED *****
00200      ;***** EXIT: ALL REGISTERS SAVED *****
00210      ;*****
00220      DRAWL:      PUSH      BC          ;SAVE REGISTERS
00230      PUSH      DE
00240      PUSH      HL
00250      PUSH      AF
00260      LD        A,C
00270      OR        A
00280      JR        NZ,DRA060
00290      ; HORIZONTAL LINE HERE
00300      POP        AF
00310      ORA050:    LD        (HL),A
00320      INC        HL
00330      DJNZ      DRA050
00340      JR        DRA090
00350      ;VERTICAL LINE HERE
00360      DRA060:    POP        AF
00370      DE,64
00380      DRA065:    LD        (HL),A
00390      AD0        HL,DE
00400      DJNZ      DRA065
00410      DRA090:    POP        HL
00420      POP        DE
00430      POP        BC
00440      RET
00450      END

```

Figure 9-9. DRAWL Subroutine

Drawing Patterns and Figures

You can also use graphics done on a character-oriented basis to draw patterns or figures. In this case, you must “plot” the figure on a display worksheet or graph paper, convert it to the proper graphics code, and then output it to the correct character positions.

One of the programming problems that seems to come up time and time again is display of “large format” characters. The Tic-Tac-Toe program of Chapter 15 uses large characters for messages, and it’s frequently done in other game programs. Let’s look at the techniques involved in this program; it’ll illustrate the general approach for displaying other types of patterns.

The first task in producing patterns such as alphanumeric characters is to draw out the patterns to be displayed. Figure 9-10 shows a typical pattern for the “large characters” used in Chapter 14. Each character is an 8 by 6 matrix. The pixels are filled in to produce a pleasing alphabetic or other character. You could use any size matrix, but a multiple of two works out conveniently for the horizontal dimension while a multiple of three works out nicely for the vertical dimension, due to the 2 by 3 **mapping** of pixels to a character position.

— Hints and Kinks 9-5 —

Large Character Format

The approach of defining a matrix of dots to represent alphanumeric and special characters is used to generate characters for video displays, dot matrix printers, and the like. Reference descriptions of “character generator” chips to find characters already defined in 5 by 7, 6 by 8, and other formats. You can then easily convert them to tables such as the one used in the Tic-Tac-Toe Program, saving you the work of drawing them out manually.

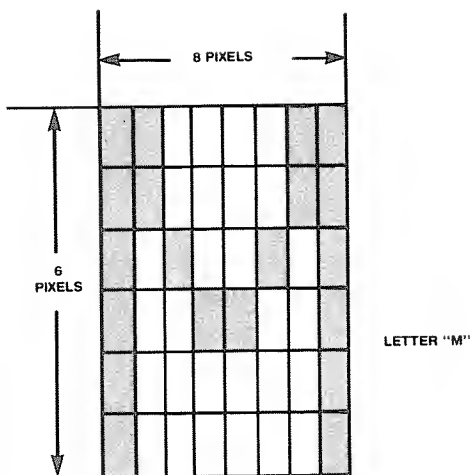


Figure 9-10. "Large" Character Pattern

After you've established the pattern, you convert it to a graphics-character data value. In this case, there are four graphics characters horizontally and two vertically (a total of eight). Each graphics character is then converted to eight bytes as shown in Figure 9-11.

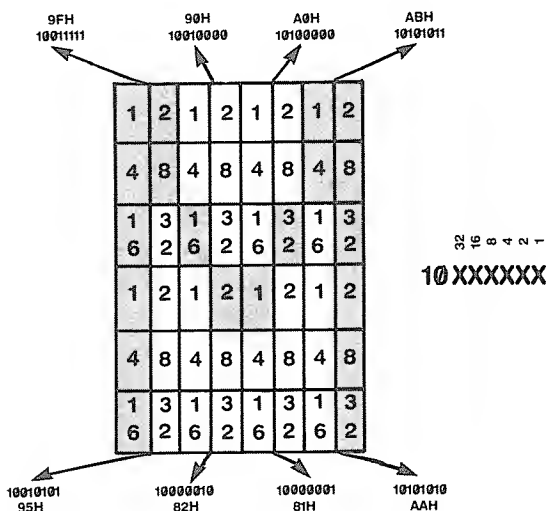


Figure 9-11. Pattern Conversion

Now the eight bytes per character are stored in a table. If the patterns have some numerical sequence, you may store them in logical fashion, such as 0 through 9. In this case, the characters were stored from A through Z, followed by space, -, ?, and !. A second table, CTAB, then holds the ASCII equivalent in the same order, as shown in Figure 9-12. To locate any character data, CTAB is first scanned for the character. When it is found, its index is multiplied by 8 to give the location of the character data in DOTTAB. **Character data in DOTTAB doesn't have the most significant bit set; it's set in the graphics driver routine (LARGEC).**

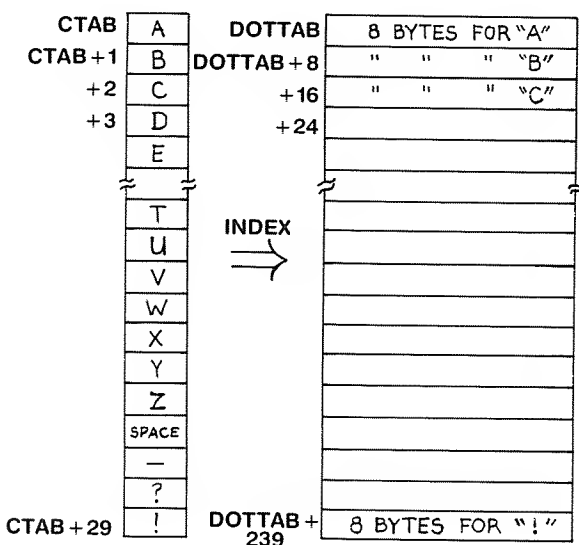


Figure 9-12. CTAB/DOTTAB Relationship

Outputting a large-format character proceeds like so:

1. Search CTAB for the character.
2. Use the CTAB index*8 to find the character data in DOTTAB.
3. Output the first four bytes from DOTTAB to a screen character position start through start + 3.

4. Output the next four bytes from DOTTAB to the screen start+64 through screen start+67.

The LARGE_C subroutine from Chapter 15 is shown in Figure 9-13. It uses the steps above to output a large character to a given screen position. A contains the character to be output in ASCII, while IY contains the screen address. In use, you would write an ASCII character string to the screen by repeated calling of LARGE_C with the next ASCII character. Between calls, the screen pointer is incremented by two to provide spacing between characters.

[illegible]

Figure 9-13.
LARGEC Routine

The MATSR subroutine within LARGE C writes out four bytes for the row. It is called twice, once with the screen start address, and once with the screen start address + 64. The remainder of the logic in the program involves searching CTAB and accessing DOTTAB.

```

0000      00100      ORG      00000H
00400      ;*****
00410      ;* SUBROUTINE TO SET OR RESET A PIXEL GIVEN X (0-127) *
00420      ;* IN H REGISTER AND Y (0-47) IN L REGISTER. *
00430      ;* (A)=0 FOR SET, 1 FOR RESET. *
00440      ;*****
00450      ;
0000 F5      00460 SETRST  PUSH  AF      ;SAVE SET/RESET FLAG
0001 5C      00470      LD      E,H      ;X
0002 7D      00480      LD      A,L      ;Y
0003 C83B    00490      SRL      E      ;GET CHAR POSITION (0-63) IN E
0005 1500    00500      LD      D,0      ;SET COL# TO 0
0007 3001    00510      JR      NC,SET10 ;GO IF COL#=0
0009 14      00520      INC      D      ;COL#=1
000A 06FF    00530 SET10  LD      B,OFFH ;-1 TO B
000C 04      00540 SET20  INC      B      ;BUMP QUOTIENT IN B=LINE#
000D D603    00550      SUB      3      ;SUCCESSIVE SUBT FOR /3
000F F20CC0  00560      JP      P,SET20 ;GO IF NOT NEGATIVE
0012 C603    00570      ADD      A,3      ;ADD BACK FOR REMAINDER=ROW#
0014 07      00580      RLCA      ;(ROW#)*2
0015 82      00590      ADD      A,D      ;(ROW#)*2+COL#-BIT POS
0016 4F      00600      LD      C,A      ;SAVE BIT POS IN C
0017 68      00610      LD      L,B      ;LINE #
0018 2500    00620      LD      H,0      ;NOW IN HL
001A 0606    00630      LD      B,6      ;SHIFT COUNT
001C 29      00640 SET30  ACD      HL,HL ;MULTIPLY LINE#*64
001D 10FD    00650      DJNZ    SET30 ;LOOP TIL DONE
001F 1600    00660      LD      D,0      ;DE NOW HAS CHAR POS
0021 19      00670      ADD      HL,DE ;(LINE#)*64+CHAR POS IN HL
0022 11003C  00680      LD      DE,3C00H ;START OF VIDEO
0025 19      00690      ADD      HL,DE ;(LINE#)*64+CHAR POS+3C00H
0026 0600    00700      LD      B,0      ;BC NOW HAS BIT POS
0028 F1      00710      PDP      AF      ;GET SET/RESET FLAG
0029 B7      00720      OR      A      ;TEST FLAG
002A 200C    00730      JR      NZ,RESET ;GO IF RESET
002C DD214AC0 00740      LD      IX,MASK ;START OF MASK TABLE
0030 DD09    00750      ADD      IX,BC ;POINT TO MASK
0032 7E      00760      LD      A,(HL) ;LOAD PIXEL
0033 DC5600  00770      OR      (IX) ;SET PIXEL
0036 77      00780 SET36  LD      (HL),A ;STORE IN VIDEO
0037 C9      00790      RET      ;RETURN
0038 DD214AC0 00800 RESET  LD      IX,MASK1 ;RESET MASK TABLE
003C DD09    00810      ADD      IX,BC ;POINT TO MASK
003E 7E      00820      LD      A,(HL) ;LOAD PIXEL
003F DDA600  00830      AND      (IX) ;RESET PIXEL
0042 18F2    00840      JR      SET36 ;GO TO STORE, RETURN
0044 81      00850 MASK   DEFB    81H ;MASK TABLE
0045 82      00860      DEFB    82H
0046 84      00870      DEFB    84H
0047 88      00880      DEFB    88H
0048 90      00890      DEFB    90H
0049 A0      00900      DEFB    0A0H
004A FE      00910 MASK1  DEFB    0FEH
004B FD      00920      DEFB    0FDH
004C FB      00930      DEFB    0FBH
004D F7      00940      DEFB    0F7H
004E EF      00950      DEFB    0EFH
004F DF      00960      DEFB    0DFH
0000      00970      END
00000 TOTAL ERRORS

```

Figure 9-14. SETRST Routine

Drawing Random Points

Now let's get back to the problem of setting and resetting random points. We know from our previous discussion how to compute the bit and byte address of the pixel, given an x,y coordinate. Figure 9-14 shows a subroutine that will set or reset any given pixel. Entry is made with X (0 through 127) in H and Y (0 through 47) in L. The A register is 0 for a set function and 1 for a reset function.

First compute the address of the byte containing the pixel by the algorithm previously described. Then use the bit position to access a "mask" table of one of six entries. There are two mask tables: one for setting a pixel; one for resetting a pixel. The byte containing the pixel is then ANDed with the mask and stored again to set or reset one of the six pixel bits in the byte location.

You should have first set all character positions that are to be used for graphics to a graphics "null" character 80H to ensure that graphics mode and not character mode is in force.

Hints and Kinks 9-6

If an 80H is Not Used . . .

If you attempt a SET/RESET for a pixel in a character position that has not been initialized by 80H, strange results may occur in an upper case TRS-80. A blank character position could have either 20H (ASCII space) or 80H (graphics null) in video memory. If it has 20H, attempting to SET a point and then turning on bit 7 results in 101X XXXX, where X is the pixel that has been set. This results in two pixels being displayed, one for the SET and one from the 20H! Always clear the area to be used for graphics with 80H before display work!

Animation

This routine takes about 400 microseconds to set or reset a pixel, making the number of pixels that can be processed per second about 2500. This is 20 times or so faster than SET/RESET in BASIC, but still somewhat slow for truly high-speed processing. If a “frame” of a picture was all 6144 pixels, for example, then only one third of a frame per picture could be processed per second. That’s also assuming that no other “number-crunching” was being done.

Of course, if the number of active pixels per frame is fewer, then we can process something close to **animation** rates of 16 frames per second. If each new frame of data RESET one half of the points and SET a number of points equal to half on the screen, we’d be SETting and RESETting the average number of points per frame, then, for each frame. If we can process 2500 points per second, that gives us about 156 points per frame, which is somewhat sparse for many pictures, but about the best that can be done. Another problem here is obtaining the **data base** for the animation. (The thought of **digitizing** 100,000 points for 40 seconds worth of Star Warp, or some other game, gives me pause . . .)

Line Drawing

We’ve discussed the simple procedures of drawing horizontal and vertical straight lines, but how about **angled lines**?

The BASIC approach to this problem would undoubtedly involve trigonometry and be very slow. However, there is a non-trigonometric approach that would work well in assembly language to permit high-speed processing. The one we’ll illustrate here involves minimum math and a somewhat different approach, so bear with us — it’ll be fairly high speed and worth the effort.

Figure 9-15 shows a line on the TRS-80 screen. It starts in the upper left and is drawn diagonally at a shallow angle. The distance from X1 to X2 is equal to some “delta X”

expressed in pixel units. In this case, X1 is 10, X2 is 20, delta X is 20-10 or 10 units. The distance from Y1 to Y2 is equal to some "delta Y," expressed in the same units. In this case delta Y is 21-15, or 6 units.

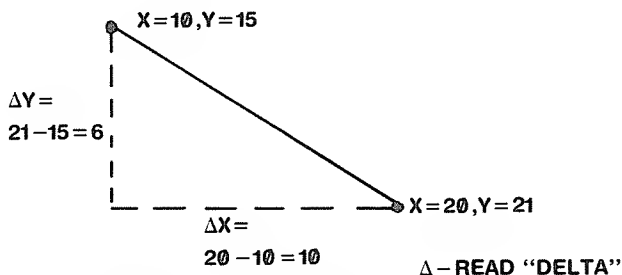


Figure 9-15. Angled Line

If we SET a point at every X value — 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20 — what points would have to be set for Y values? We can easily find out by computing the amount that Y has to be incremented for every new X position. This is:

$$\text{DELTA Y/DELTA X} = 6/10 \text{ PIXEL}$$

In other words, every time we increment X by 1, we increment Y by .6. The points that would be set for this line would therefore be:

X	Y	X	Y
10	15	16	18.6
11	15.6	17	19.2
12	16.2	18	19.8
13	16.8	19	20.4
14	17.4	20	21
15	18.0		

Notice that we wrote some points more than once, but at least the algorithm is straightforward.

There is a minor problem here — how do we work with fractions in assembly language? In BASIC we have the mechanism built into a FOR... TO... STEP loop. How do we implement it here?

Scaling

We'll **scale** Y upwards by 256. This means we'll hold Y as a number multiplied by 256. The high-order 8 bits will be the integer portion of the number, and the low-order 8 bits will be the fractional part, as shown in Figure 9-16. When we find $\Delta Y / \Delta X$, we'll get a result that is actually $(\Delta Y * 256) / X$ as shown in Figure 9-17. We can then add the result to the 16-bit scaled representation of Y to get each new increment of Y. After every add, we'll take the new value of X and the **integer** (higher 8 bits) of Y to set the new point. The whole process for the points we've been discussing is shown in Figure 9-18.



Figure 9-16. Scaling Example

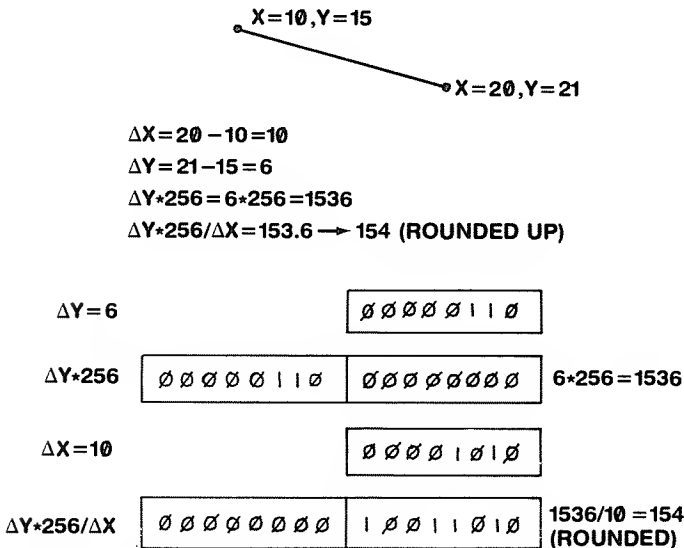
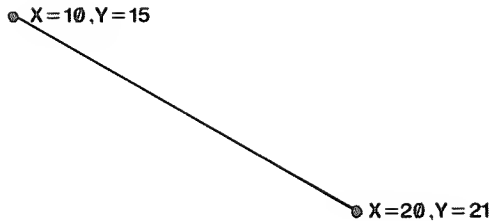


Figure 9-17. Scaling With Division



$$\begin{aligned}\Delta X &= 20 - 10 = 10 \\ \Delta Y &= 21 - 15 = 6 \\ \Delta Y * 256 &= 6 * 256 = 1536 \\ \Delta Y * 256 / \Delta X &= 153.6 \rightarrow 154 \text{ (ROUNDED)}\end{aligned}$$

X	Y	16-BIT SCALED VALUE	Y INTEGER
10	00001111	15 * 256 = 3840	15
11	00001111	3840 + 154 = 3994	15
12	00010000	3994 + 154 = 4148	16
13	00010000	4148 + 154 = 4302	16
14	00010001	4302 + 154 = 4456	17
15	00010010	4456 + 154 = 4610	18
16	00010100	4610 + 154 = 4764	18
17	00010110	4764 + 154 = 4918	19
18	00010111	4918 + 154 = 5072	19
19	00010101	5072 + 154 = 5226	20
20	00010110	5226 + 154 = 5380	21

Figure 9-18. Basic Line Drawing Algorithm

Hints and Kinks 9-7 Scaling

The problem of scaling was not a trivial one in early computer work. (*Digital Computer Programming* by McCracken, 1957, devotes a chapter to 'Decimal Point Location Methods.') Higher-level languages solved the problem of working with mixed numbers, and most 'number crunching' applications use a language other than assembly language to avoid laborious scaling processing.

Now this algorithm works fine for the one case we've been discussing, but how about the **general case**? When the

angle is more acute, as shown in Figure 9-19, the situation changes. Now delta Y is greater than delta X, and we must increment Y by one and X by some fractional value. Very well — for this case we'll do just that. But wait a minute — how many cases are there?

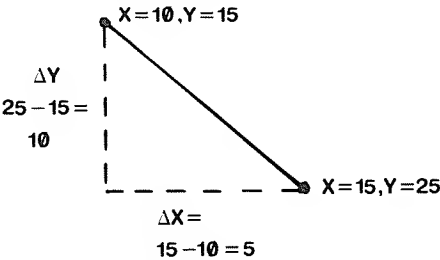


Figure 9-19. Second Case of Angled Line

There are actually eight cases, when we consider the size of delta X to delta Y, and the direction of the line. They're shown in Figure 9-20. Four of these increment X, and four increment Y.

ΔX	ΔY	$ \Delta X : \Delta Y $	LINE	X INCREMENT	Y INCREMENT
+	+	+		+1	$\Delta Y/\Delta X$
+	+	-		$\Delta X/\Delta Y$	+1
+	-	+		+1	$\Delta Y/\Delta X$
+	-	-		$\Delta X/\Delta Y$	-1
-	+	+		-1	$\Delta Y/\Delta X$
-	+	-		$\Delta X/\Delta Y$	+1
-	-	+		-1	$\Delta Y/\Delta X$
-	-	-		$\Delta X/\Delta Y$	-1

- NOTES:
1. ΔX IS $X_2 - X_1$
 2. ΔY IS $Y_2 - Y_1$
 3. $|\Delta X|$ IS ABSOLUTE VALUE
 4. $|\Delta Y|$ IS ABSOLUTE VALUE
 5. $|\Delta X|:|\Delta Y|$ IS $|\Delta X|$ COMPARED TO $|\Delta Y|$. IF +, $|\Delta X|$ IS LARGER THAN $|\Delta Y|$. IF -, $|\Delta Y|$ IS LARGER THAN $|\Delta X|$
 6. INCREMENTS SHOW WHICH VARIABLE WILL BE STEPPED ONE UNIT AT A TIME, AND WHICH WILL BE FRACTIONALLY STEPPED

Figure 9-20. Line Drawing Configurations

The algorithm for the LINE subroutine goes like this:

1. Find delta X by subtracting $X_2 - X_1$. Call this value DX.
2. Find delta Y by subtracting $Y_2 - Y_1$. Call this value DY.

3. If the absolute value of DX is greater than the absolute value of DY, PUT $1*256$ in XI (X increment), absolute value of DX+1 in CT (count), and absolute value of $DY*256/DX$ in YI (Y increment). X will vary in steps of one in this case.
4. If the absolute value of DX is less than or equal to the absolute value of DY, put $1*256$ in YI, absolute value of DY+1 in CT, and absolute value of $DX*256/DY$ in XI. Y will vary in steps of one in this case.
5. If DX is negative, negate XI.
6. If DY is negative, negate YI.
7. Increment X and Y from their starting values for a number of steps equal to CT.

This algorithm (with some nuances) is shown in the BASIC subroutine of Figure 9-21. It does all of the number crunching through step 4 above. You could implement the code in assembly language, but my trial effort was about 120 lines (!). The parameters from the BASIC processing are POKEd into locations FF00H through FF0AH for use by the assembly-language LINE program that does the high-speed line drawing. In this way you have the best of both BASIC and assembly language (and I can avoid explaining 120 lines of code!)

```

10000 DX=X2-X1
10010 DY=Y2-Y1
10020 IF (DX=0) AND (DY=0) THEN XI=0:YI=0:CT=1:GOTO 10070
10030 IF ABS(DX)>ABS(DY) THEN XI=256:CT=ABS(DX)+1:YI=ABS(DY*256/DX)
10040 IF ABS(DX)<=ABS(DY) THEN YI=256:CT=ABS(DY)+1:XI=ABS(DX*256/DY)
10050 IF ABS(DX)<>(CT-1)*XI/256 THEN XI=XI+1
10060 IF ABS(DY)<>(CT-1)*YI/256 THEN YI=YI+1
10070 IF DX<0 THEN POKE &HFF09,1 ELSE POKE &HFF09,0
10080 IF DY<0 THEN POKE &HFF0A,1 ELSE POKE &HFF0A,0
10090 X=X1*256
10100 Y=Y1*256
10110 POKE &HFF00,X-INT(X/256)*256
10120 POKE &HFF01,INT(X/256)
10130 POKE &HFF02,Y-INT(Y/256)*256
10140 POKE &HFF03,INT(Y/256)
10150 POKE &HFF04,XI-INT(XI/256)*256
10160 POKE &HFF05,INT(XI/256)
10170 POKE &HFF06,YI-INT(YI/256)*256
10180 POKE &HFF07,INT(YI/256)
10190 POKE &HFF08,CT
10200 DEFUSR0=&HFF0B
10210 A=USR0(0)
10220 RETURN

```

**Figure 9-21. Line Routine in
BASIC**

The LINE routine is shown in Figure 9-22. It performs steps 5 through 7 of the algorithm. LINE calls the SETRST subroutine described previously. LINE will draw a typical 20-point line in about 10 milliseconds, not including the BASIC processing portion. It will also draw vertical or horizontal lines and points, but not as efficiently as the other techniques described.

```

FF00      00100      ORG      OFF00H
00110      ;*****
00120      ; LINE SUBROUTINE. DRAWS A STRAIGHT LINE BETWEEN ANY
00130      ; TWO GIVEN POINTS. OPERATES IN CONJUNCTION WITH BASIC
00140      ; DRIVER PROGRAM. ENTER WITH BLOCK SETUP AS FOLLOWS:
00150      ; BLOCK+0,+1: SCALED X VALUE/+2,+3: SCALED Y VALUE/
00160      ; +4,+5: ABSOLUTE X INCREMENT, SCALED/+6,+7: ABSOLUTE
00170      ; Y INCREMENT, SCALED/+8:COUNT/+9: 1 IF NEGATE X INC
00180      ; ELSE 0/+A: 1 IF NEGATE Y INC ELSE 0
00190      ;*****
00200      ;
000B      00210 BLOCK DEFS      11
FF0B      3A09FF      LD      A,(BLOCK+9)      ;GET INCREMENT SENSE
FF0E      B7          00230      OR      A      ;TEST
FF0F      280D      00240      JR      Z,LIN010      ;GO IF XINC +
FF11      210000      LD      HL,0      ;ZERO HL
FF14      ED5B04FF      LD      DE,(BLOCK+4)      ;GET XINC
FF18      B7          00270      OR      A      ;ZERO C
FF19      ED52      00280      SBC      HL,DE      ;NEGATE
FF1B      2204FF      LD      (BLOCK+4),HL      ;STORE - XINC
FF1E      3A0AFF      00300      LD      A,(BLOCK+10)      ;GET INCREMENT SENSE
FF21      B7          00310      OR      A      ;TEST
FF22      280D      00320      JR      Z,LIN020      ;GO IF YINC +
FF24      210000      LD      HL,0      ;ZERO HL
FF27      ED5B06FF      LD      DE,(BLOCK+6)      ;GET YINC
FF2B      B7          00350      OR      A      ;ZERO C
FF2C      ED52      00360      SBC      HL,DE      ;NEGATE
FF2E      2206FF      LD      (BLOCK+6),HL      ;STORE - YINC
FF31      DD2100FF      LD      IX,BLOCK      ;POINT TO BLOCK
FF35      DD6601      00390      LD      H,(IX+1)      ;GET X
FF38      DD6E03      00400      LD      L,(IX+3)      ;GET Y
FF3B      AF          00410      XOR      A      ;FOR SET
FF3C      CD5FFF      00420      CALL      SETRST      ;SET POINT
FF3F      2A00FF      00430      LD      HL,(BLOCK)      ;CURRENT X
FF42      ED5B04FF      00440      LD      DE,(BLOCK+4)      ;INCREMENT
FF46      19          00450      ADD      HL,DE      ;BUMP
FF47      2200FF      00460      LD      (BLOCK),HL      ;STORE
FF4A      2A02FF      00470      LD      HL,(BLOCK+2)      ;CURRENT Y
FF4D      ED5B06FF      00480      LD      DE,(BLOCK+6)      ;INCREMENT
FF51      19          00490      ADD      HL,DE      ;BUMP
FF52      2202FF      00500      LD      (BLOCK+2),HL      ;STORE
FF55      3A08FF      00510      LD      A,(BLOCK+8)      ;COUNT
FF58      3D          00520      DEC      A      ;DECREMENT
FF59      3208FF      00530      LD      (BLOCK+8),A      ;SAVE
FF5C      20D3      00540      JR      NZ,LIN020      ;GO IF MORE
FF5E      C9          00550      RET      ;RETURN TO BASIC
00560      ;*****
00570      ; SUBROUTINE TO SET OR RESET A PIXEL GIVEN X (0-127)
00580      ; IN H REGISTER AND Y (0-47) IN L REGISTER.
00590      ; (A)=0 FOR SET, 1 FOR RESET.
00600      ;*****
00610      ;
FF5F      F5          00620      SETRST      PUSH      AF      ;SAVE SET/RESET FLAG
FF60      5C          00630      LD      E,H      ;X
FF61      7D          00640      LD      A,L      ;Y
FF62      CB3B      00650      SRL      E      ;GET CHAR POSITION (0-63) IN E
FF64      1600      00660      LD      D,0      ;SET COL# TO 0
FF66      3001      00670      JR      NC,SET10      ;GO IF COL# = 0
FF68      14          00680      INC      D      ;COL# = 1
FF69      06FF      00690      SET10      LD      B,OFFH      ;-1 TO B
FF6B      04          00700      SET20      INC      B      ;BUMP QUOTIENT IN B=LINE#

```


FF6C D6D3	DD71D	SUB	3	;SUCCESSIVE SUBT FOR /3
FF6E F26BFF	0072D	JP	P,SET2D	;GD IF NOT NEGATIVE
FF71 C603	D073D	ADD	A,3	;ADD BACK FOR REMAINDER=ROW#
FF73 07	00740	RLCA		;(ROW#)*2
FF74 82	0075D	ADD	A,D	;(RDW#)*2+COL#=BIT PDS
FF75 4F	D076D	LD	C,A	;SAVE BIT POS IN C
FF76 68	0077D	LD	L,B	;LINE #
FF77 2600	00780	LD	H,0	;NOW IN HL
FF79 D6D6	DD79D	LD	B,6	;SHIFT COUNT
FF7B 29	DD8DD	ADD	HL,HL	;MULTIPLY LINE#*64
FF7C 10FD	0081D	DJNZ	SET30	;LDOP TIL DONE
FF7E 16DD	DD82D	LD	D,0	;DE NOW HAS CHAR PDS
FF8D 19	DD83D	ADD	HL,DE	;(LINE#)*64+CHAR PDS IN HL
FF81 11DD3C	DD84D	LD	DE,3CDDH	;START OF VIDEO
FF84 19	00850	ADD	HL,DE	;(LINE#)*64+CHAR POS+3C00H
FF85 D6DD	DD86D	LD	B,0	;BC NOW HAS BIT POS
FF87 F1	DD87D	POP	AF	;GET SET/RESET FLAG
FF88 B7	DD88D	OR	A	;TEST FLAG
FF89 2D0C	0089D	JR	NZ,RESET	;GO IF RESET
FF8B DD21A3FF	DD9D0	LD	IX,MASK	;START DF MASK TABLE
FF8F DDD9	DD91D	ADD	IX,BC	;POINT TD MASK
FF91 7E	DD92D	LD	A,(HL)	;LDAD PIXEL
FF92 DDB6DD	DD930	OR	(IX)	;SET PIXEL
FF95 77	DD94D	LD	(HL),A	;STORE IN VIDEO
FF96 C9	DD95D	RET		;RETURN
FF97 DD21A9FF	DD960	RESET	LD	IX,MASK1
FF9B DDD9	0097D	ADD	IX,BC	;RESET MASK TABLE
FF9D 7E	D0980	LD	A,(HL)	;POINT TO MASK
FF9E DDA6DD	DD99D	AND	(IX)	;LOAD PIXEL
FFA1 18F2	D1DDD	JR	SET36	;RESET PIXEL
FFA3 81	0101D	MASK	DEFB	81H
FFA4 82	D1D2D		DEFB	82H
FFA5 84	D1D3D		DEFB	84H
FFA6 88	D104D		DEFB	88H
FFA7 9D	D105D		DEFB	90H
FFA8 AD	D1D6D		DEFB	DADH
FFA9 FE	D107D	MASK1	DEFB	DFEH
FFAA FD	D1D80		DEFB	DFDH
FFAB FB	D1D90		DEFB	OFBH
FFAC F7	D11DD		DEFB	DF7H
FFAD EF	D111D		DEFB	DEFH
FFAE DF	01120		DEFB	DDFH
00D0	01130	END		
D0DD0	TOTAL ERRDRS			

Figure 9-22. LINE Routine

Chapter Ten

Cassette Output, Music, and Parallel Printers

We'll be looking at two types of input/output processing in this chapter: I/O on the cassette **port** and **parallel** printer output. We'll also take a look at system I/O in general.

The TRS-80 reads and writes cassette tape primarily using software drivers rather than hardware logic. You can use these software drivers in ROM to read and write down to a bit at a time from cassette. Because the cassette port is easily addressable at an assembly-language level, you can also use it to generate "square wave" outputs that can be musical tones or other signals.

System parallel printers are addressed differently than the cassette port. We'll see how simple **printer drivers** may be coded.

Input/Output Programming

Many programmers are unnecessarily upset by input/output programming. Part of the reason for this is that I/O is done automatically in large computer systems; the programmer must make special supervisor calls to perform I/O through the operating system used on the system. This approach is necessary because the system is performing many tasks (many "job runs") simultaneously and is increasing the **throughput** by overlapping processing on one job with I/O on another. Another reason for the mysteries of I/O is some programmers feel that it requires an understanding of (ugh!) hardware.

In the TRS-80, we're operating in a different environment than a large multi-programming system (and believe it or

not, a **much more** efficient environment on an individual basis). We can perform I/O ourselves without having to go through the operating system; of course, the provision is also there to have the operating system do our work for us. It's user's choice. As far as knowing "hardware," most programmers don't realize the simplicity of the logic involved in interfacing to most computer peripherals. We'll show you how easy it is in this chapter.

Z-80 and TRS-80 Input/Output

There are two basic types of I/O in any microcomputer system — I/O mapped I/O and memory-mapped I/O.

I/O Mapped I/O

I/O Mapped I/O uses the I/O instructions in the Z-80. There are four that we'll talk about here: `IN A,(n)`; `IN r,(C)`; `OUT (n),A`; and `OUT (C),r`. You can accomplish the same thing with all of them: transfer one byte of data between an external I/O device and a CPU register. The `IN` instructions read one byte from the external device into a CPU register, while the `OUT` instructions write one byte from a CPU register to an external device.

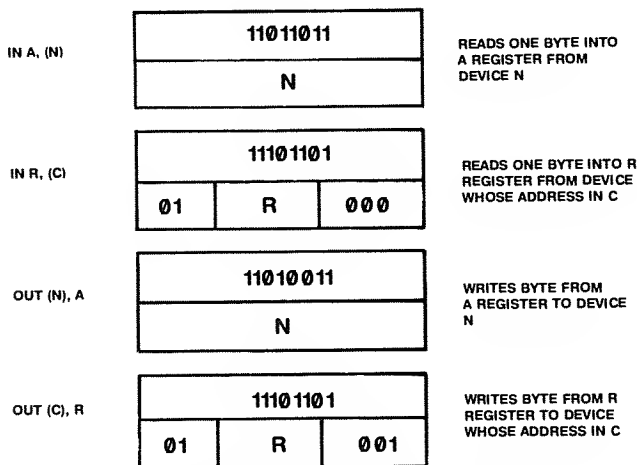
—Hints and Kinks 10-1—

Other I/O Instructions

The other I/O instructions are the "block" I/O instructions. They are somewhat similar in action to the block move instructions. Data can be transferred between memory and an I/O device in a block by setting up a block address in HL and a byte count of 1-256 in the B register; the C register holds the address of the I/O device.

Like the block moves, the I/O block instructions may go forward through the block (`INIR,OTIR`), backward through the block (`INDR,OTDR`), or "semi-automatically" (external loop back to `INI, IND, OUTI, or OUTD`). The "automatic" I/O block instructions cannot be used unless the I/O controller has been designed for such a transfer.

The format of these instructions is shown in Figure 10-1. All use an **I/O address**. This is an eight-bit address that is contained in the I/O instruction itself (IN A,(n);OUT (n),A) or is in the C register (IN r,(C);OUT (C),r). The I/O address can be 0 through 255.



**Figure 10-1. I/O Instruction
Formats**

When an I/O instruction is executed, it goes through a predefined sequence. The I/O address is first sent out along the system address lines A7-A0. Shortly afterwards, data is output along the data lines D7-D0 for the OUT, or input from the data lines D7-D0 for the IN.

External I/O devices are designed to expect this sequence. If the I/O device senses that an I/O instruction is being executed (there is another signal line that performs this function), it reads the address lines to determine if it's being addressed. If it is, it either reads in the byte of data from the data lines or sends back a byte of data to the data lines.

Hints and Kinks 10-2
I/O Mapped I/O Signals

The actual sequence for I/O mapped I/O goes like this: The usual I/O cycle is three cycles: T1, T2, and T3. The 'port address' is put on address lines A0-A7 during T1. Next, at about T2, the TRS-80 OUT* or IN* signal (one or the other) goes to zero. Each is on a separate line. IN* signifies that an IN instruction is being performed, and OUT* denotes an OUT instruction.

If an input is being done and if an external device's address is on the address bus, it responds by placing a byte of data on the data bus lines D7-D0. The byte is input to the CPU register at about T3.

If an output is being done and if an external device is being addressed, the device 'strokes in' the data byte on D7-D0 in T2 or T3. The data was placed on the data lines sometime in T1.

Since there may be 256 separate **port addresses** (I/O addresses in the I/O instructions), there may be as many as 256 separate I/O devices, all looking for their address on the address lines so that they can perform their built in function of reading or writing one byte of data. In practice, there are probably only one or two devices hooked up to any microcomputer system.

Most TRS-80 Model I systems have only two devices. One is the cassette read/write logic (address 0FFH), and the other is the RS-232-C interface board (addresses 0E8H through 0EBH). Although these are somewhat integrated into the TRS-80 system, they are truly external devices viewed from the standpoint of the Z-80 microprocessor.

Model III systems also use cassette and RS-232-C port addresses, but use a number of other ports for disk and system operations.

Memory-Mapped I/O

The second type of I/O that we can have in the TRS-80 is memory-mapped I/O. Here, the external device is still looking for an address on the address lines, but all address lines A15-A0 are used. In this case, the external device does not look for a signal that says "I/O instruction being executed" together with its address, but simply looks for its address. As 16 address lines are being used, 65536 separate addresses could be employed.

Hints and Kinks 10-3 Memory-Mapped I/O Signals

As in I/O mapped I/O, this sequence usually takes place in 3 T cycles. First of all, the CPU puts the device address onto the address bus lines A15-A0 during the T1 cycle. If the device is being addressed, it now looks at signals WR* and RD*.

If an input is being done, signal RD* goes low (0) during T2. An external device responds to its address and RD* by placing a byte of data on data bus line D7-D0. If an output is being done, signal WR* goes low during T2. An external device responds to a WR* and address by 'strobing in' the data on lines D7-D0. The data was placed on these lines by the CPU sometime in T1.

Note that in this type of addressing, the WR* and RD* signals are generated by instructions such as LD (HL),A for an output (WR*) and LD (nn),A for an input (RD*). The I/O device looks identical to a memory location in this mode.

The catch in the above is that some of the addresses are also used for memory! In this type of I/O, there must be a decision by the system designer on how to divide up the 64K worth of addresses into memory and I/O addresses. This division in the TRS-80 is shown in Figure 10-2, which shows the memory mapping for the TRS-80.

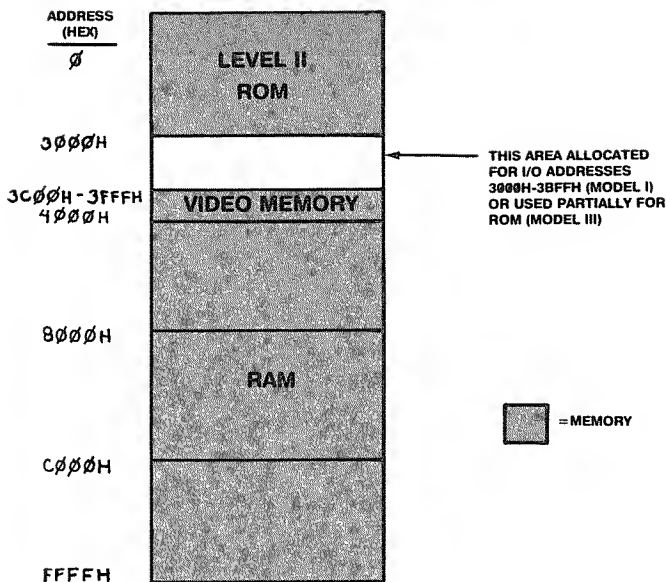


Figure 10-2. TRS-80 Memory Mapping For I/O

Of course, as we know, many of the addresses in the TRS-80 are devoted to video memory (addresses 3C00H through 3FFFH). In fact, this is very similar to regular RAM memory. Other memory-mapped addresses, however, are the system line printer (37E8H), and (in the Model I) the **disk controller chip** (37E0H, 37ECH through 37EFH), and the cassette latch in the expansion interface (37E4H).

To address any of these addresses on a read, we simply perform any **memory reference** instruction, such as LD A,(37E8H) or LD B.(HL). To address any of the addresses on a write, a store is done as in LD (37E8H),A or LD (HL),C. The memory reference instruction, of course, transfers one byte of data between the external (to the Z-80) I/O device and a CPU register.

Just about all I/O operations in the TRS-80 are performed one byte at a time by using either an I/O instruction in I/O mapped I/O, or by using a memory reference instruction in memory-mapped I/O. (Even disk operations are performed one byte at a time.)

We haven't talked about several other possible types of I/O in the TRS-80 for a good reason - they are not used in standard TRS-80 devices. However, let's mention them anyway, as the provision is there in the TRS-80 as far as bus signals.

The first of these is DMA, direct memory access. In this type of I/O, 8-bit I/O transfers are made directly to and from memory by the I/O device controller, bypassing CPU registers. During each transfer, the CPU is 'locked up' and the device controller uses the bus lines in a 'cycle-stealing' mode. This type of I/O is common for very high-speed devices where a software loop can not furnish the data at high enough rates.

The second type of I/O is the interrupt-driven I/O. This type of I/O is usually used with slow-speed devices. Each byte (for example, a keypush) generates an interrupt to the CPU, which causes an interrupt-processing routine to be entered. The interrupt-processing routine contains normal I/O instructions to read in or write out the data. The advantage of this type of I/O is that normal processing can be maintained until the interrupt occurs without any polling or overhead in testing the I/O device for the next byte.

Parallel Printer Operation

The system parallel printer uses memory-mapped I/O with an address of 37E8H. The expansion interface in the Model I or printer logic in the Model III **decodes** this address and passes data either to the printer or from the printer, as shown in Figure 10-3.

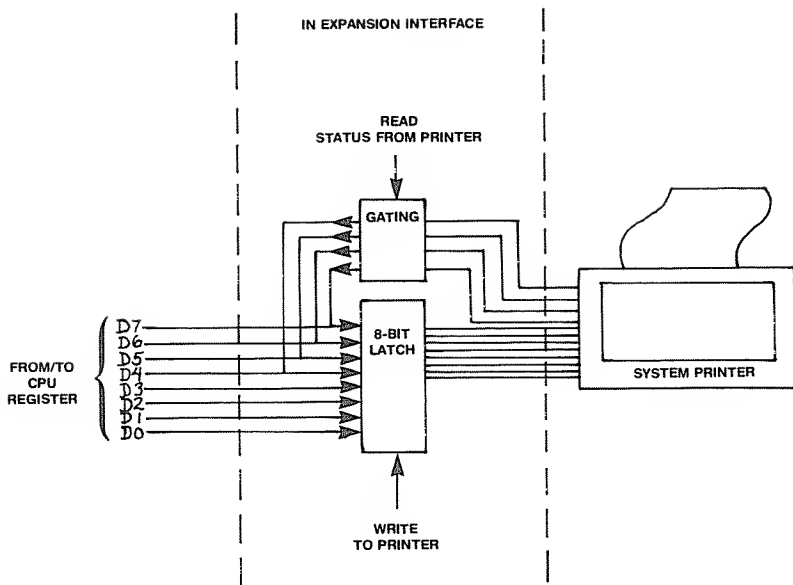


Figure 10-3. Parallel Printer Logic

Printers used on the TRS-80 use a somewhat standard **interfacing** specification called the **Centronics bus**. This specification defines the set of lines used to transfer the data and the signals for **handshaking**, timing, and signal levels.

The handshaking logic for parallel printers goes something like this. The CPU performs a memory-reference read instruction to read a byte of data from the printer **controller** logic. This byte is a **status** byte, as shown in Figure 10-4.

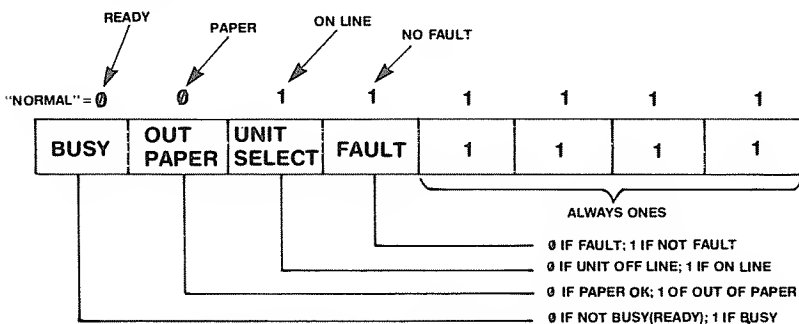


Figure 10-4. Printer Status Byte

The status byte normally contains several bits indicating "fault" conditions for the printer — printer out of paper, off-line, etc. It also contains a bit known as a **ready** bit.

The ready bit signifies that the printer is ready to accept the next byte of data to be printed. In an **unbuffered** printer, the printer is **busy** (not ready) during the time the character is being printed, and the ready bit is not set. After the character has been printed, the ready bit is set by the printer electronics. This situation is shown in Figure 10-5.

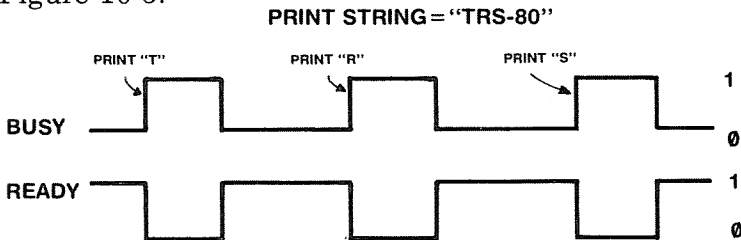


Figure 10-5. Ready/Busy Status

In a **buffered** line printer, the printer may receive the next byte of data to be printed while the printer is in the process of printing. The character is stored in a buffer memory within the printer. Of course, if the printer is printing at a rate of 100 characters per second and the program is outputting characters at rates of 50,000 characters per second, it doesn't take too long for the buffer to become filled up. In this case, the ready bit is reset until one or more characters have been read out of the buffer and printed. Almost all current line printers are of the buffered type.

Figure 10-6 shows a typical line printer **driver** program from the MORG program of Chapter 14. The status is first read from the printer controller logic by a LD A,(37E8H). The status bits are ANDed with 0F0H to mask out the bits from the other bit positions; they contain ones as they connect to nothing for a read from the printer. The other four bits are BUSY, OUTPAPER, UNIT SELECT, and FAULT. If the result is other than 30H, the printer is busy or has a fault condition, and a jump is made back to the read status instruction.

Hints and Kinks 10-5

Other Line Printer Characters

If the line printer is very complex, there may be a variety of 'control codes' with special meanings that can be output. First of all, there are carriage returns (ODH) and line feeds (OAH). Most printers also have a 'top of form,' which advances the paper to the next page (OCH).

Another common control code is the 'BEL' code (07H) which literally rang a bell on early teleprinters; on modern line printers it usually sounds an electronic alarm.

Depending upon the printer, you may have codes for such things as underlining, setting vertical spacing, and setting horizontal print density.

On character-oriented printers - such as the Diablo, Qume, and NEC - you have a whole set of special 'escape sequences' that control such things as horizontal and vertical spacing in fractions of an inch, vertical and horizontal tabs, and ribbon selection. These sequences are not single codes but are a series of characters, many times started by an 'escape' character (1BH). If these printers are to be utilized to their fullest advantage, the printer software driver must make provision for outputting such sequences; this may increase the task of writing an assembly-language driver by a factor of ten or more!

```

05390 :
05400 ; LINE PRINTER STATUS AND PRINT CHARACTER SUBROUTINE
05410 :
83AF F5      05420 LPSTAT  PUSH    AF          ;SAVE CHAR
83B0 3AE837  05430 LPS010  LD      A,(37E8H)    ;GET STATUS
83B3 E6F0    05440        AND      0F0H        ;MASK OUT GARBAGE BITS
83B5 FE30    05450        CP       30H         ;TEST FOR BUSY
83B7 20F7    05460        JR       NZ,LPS010    ;GO IF BUSY
83B9 F1      05470        POP      AF          ;RESTORE CHAR
83BA 32E837  05480        LD       (37E8H),A    ;OUTPUT
83BD C9      05490        RET                ;RETURN

```

Figure 10-6. Line Printer Driver Program

If the line printer is ready, the ASCII character is retrieved from the stack and output to address 37E8H by an LD instruction. All data output to the printer must be ASCII data, except for possible special control codes unique to the printer. (A BEL code of 07 would sound an alarm, or a code of 29 might select 20 characters per inch spacing, for example.)

Of course, the line printer driver shown above is the lowest-level subroutine for communication with a system printer. There may be others above it that control message output, format lines, and so forth. This simple **protocol** is typical of many peripheral devices such as line printers, paper tape readers, and card readers.

Cassette I/O

The cassette **controller** is contained in the CPU logic. Like the printer controller, it contains logic to decode its address and to transfer data between a CPU register and the I/O device. While the printer controller uses memory-mapped I/O addressing, however, the cassette logic uses I/O mapped I/O via IN and OUT instructions. The cassette logic is shown in Figure 10-7.

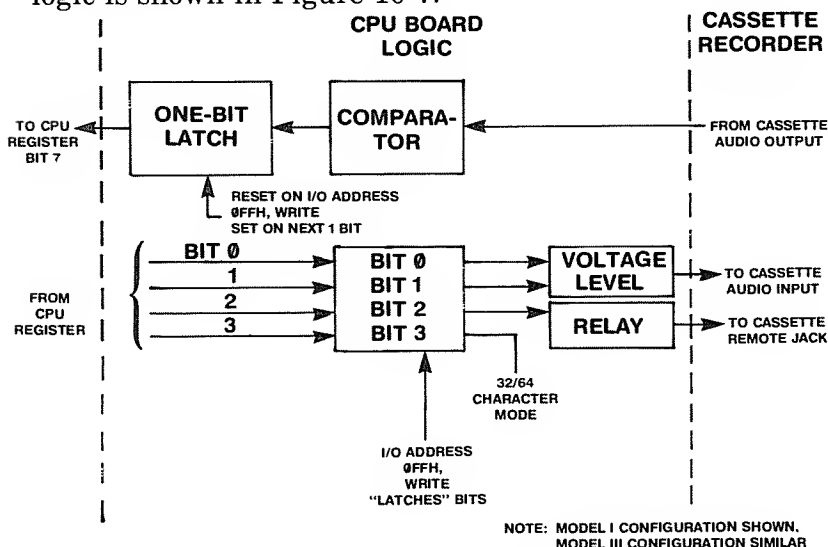


Figure 10-7. Cassette Logic

Cassette Output

When an OUT (OFFH),A or similar instruction is done, the four least significant bits in A (or another CPU register) are transferred to the cassette **latch** shown in Figure 10-7 for the Model I. The latch is really a four-bit memory cell that retains the data until rewritten. The logic for the Model III is similar, but not identical.

Bit 3 of the latch (Model I) controls the 32/64 character mode for the display. Bit 2 of the latch (Model I) controls the cassette relay that would normally be used to turn the cassette motor on and off. Bits 1 and 0 (Model I and III) generate a signal level to the cassette input for writing on cassette. Three signal levels are possible, as shown in Figure 10-8.

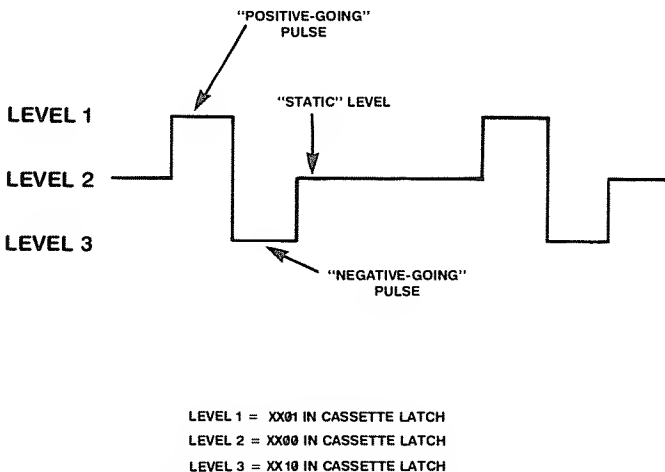


Figure 10-8. Cassette Output Levels

To write on cassette tape, a series of pulses are generated as shown in Figure 10-9. The separation between the pulses is 1 millisecond. The width of each pulse is 250 microseconds, divided into 125 microsecond segments.

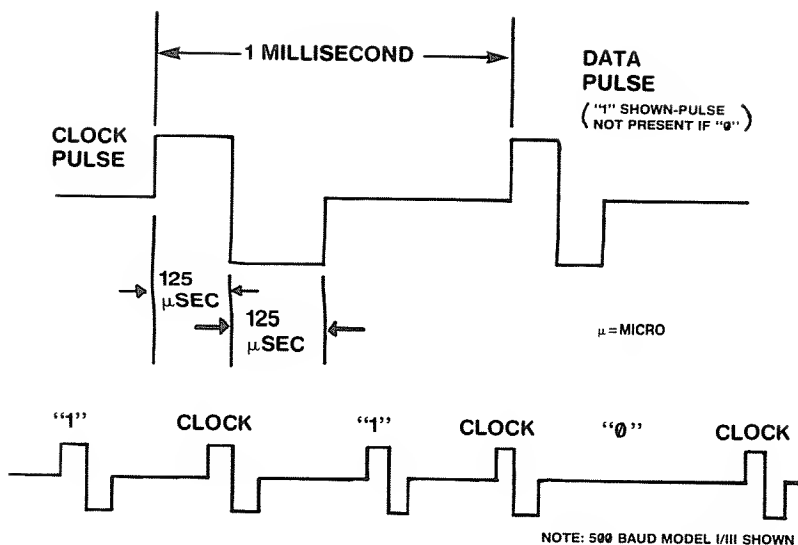


Figure 10-9. Cassette Pulse Formats

There are two pulses for every data bit. Each set of two is divided into a **clocking pulse** and a **data pulse**. There is always a clocking pulse. If there is a data pulse at the 1 millisecond time, a "1" bit is generated. If there is **no data** pulse, a "0" bit is produced, as shown in the figure.

Prior to the generation of pulses, of course, the cassette must have been turned on by a 0000X100 output, which turns on the cassette relay; the X represents the current state of 32/64 character mode (Model I), which must be retained. The positive portion of the cassette is produced by outputting a binary 0000X101, and the negative portion by a binary 0000X110. The reference level is produced by an output of 0000X100.

The normal sequence for writing to cassette is to turn on the motor, write a string of 255 bytes of zeroes (2040 bits of zeroes), and then write a **sync** byte of 0A5H. This really amounts to turning on the motor and then writing zeroes for 2040×2 milliseconds = 4.08 seconds, following with a sync byte. The sync byte is detected in the software read cassette routine and marks the start of all data (and the end of the zero header).

Cassette Input

The cassette read logic consists of a one-bit read latch and comparator logic. The software read cassette routine reads one bit from the cassette at a time. This bit is read into bit 7 by performing an `IN A,(0FFH)` or similar instruction. If the bit is a zero, there is no pulse present from the cassette input; if the bit is a one, there is a pulse present.

The following discussion applies to 500-baud cassette rates in the Model I, but is very similar for Model III operations.

The normal read cassette operation begins with turning the cassette motor on. The cassette read latch is cleared by an `IN (0FFH),A` (A may have any data). Next a series of `IN A,(0FFH)` instructions is executed until a one bit is read. This is the start of the clocking pulse. Then a delay about 500 microseconds. This puts the cassette tape past the clock pulse. Then the cassette latch is reset by an `IN (0FFH),A`. Now another 850 microseconds delay occurs. We're now positioned past where the data pulse should have been. If a data bit had been present, the read latch would now be set. If there is a one after reading the latch by an `IN A,(0FFH)`, the data bit was a one, otherwise the data bit was a zero. After this sequence, the entire process is repeated for the next data bit. Figure 10-10 shows the operation.

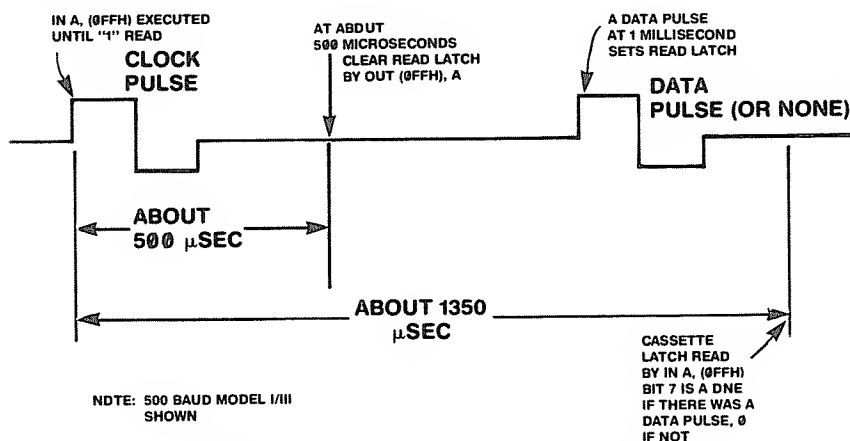


Figure 10-10. Cassette Read Timing

The first data bit to be read will be bit 7 of the 0A5H sync byte (or noise). After the entire 8 bits of the sync byte have been read and verified, the cassette software driver is "synced" to the data and can assemble the bytes of the cassette record from the individual reads.

It's entirely possible to write your own cassette I/O routines from the ground up, addressing the cassette write and read latches. However, there are ROM calls available in the EDTASM manual that you may use to perform the next level of byte-oriented operations. The calls are:

- Define drive and turn on motor
- Turn off motor
- Write leader and sync byte
- Write byte
- Read leader and sync byte
- Read byte

—Hints and Kinks 10-6 —
High-Speed Tape Operations

Since the cassette tape is primarily software driven, is it possible for you to speed up the cassette data transfer rate? The answer is in the qualifier 'primarily.' It is certainly possible to write assembly-language code to write and read cassette at 1000 baud or higher. However, since these frequencies have an entirely different set of electrical characteristics and the cassette electronics are designed to work well with the standard baud rates, a higher-speed cassette driver would be a dicey thing at best. Best to invest in that disk than in an experimental 19,200 baud cassette tape driver!

Define Drive, Motor On, Motor Off

A CALL to location 212H with the A register containing a 0 or 1 selects cassette 0 or 1 and turns on the motor. A CALL to location 01F8H deselects the cassette and turns the motor off.

Read Cassette

A CALL to location 0296H looks for the leader of 255 zeroes and the sync byte of 0A5H. It returns after a sync byte has been read. If no sync byte is read, it "hangs," looking for the elusive 0A5H. After this call has been made, a CALL to 0235H will read in the remaining bytes in the cassette record. A byte is returned in the A register.

Write Cassette

A CALL to location 0287H writes the leader and sync byte. A CALL to location 0264H with a data byte in A writes one byte of data.

The above routines may be used in any fashion to create your own tape data formats or to work with existing data formats. All tape operations will be at 500 baud (500 data bits per second) rates. For example, the size of records is normally less than 256 bytes. You may construct your own assembly-language routines to **block** more than one logical record into one physical record.

Figure 10-11 shows an assembly-language routine to write and read video display data in 1024-byte records by utilizing the ROM calls above. This is a relocatable program and is incorporated into the Level II BASIC program shown in Figure 10-12.

```

7F00      00100      ORG      7F00H
00110 : *****
00120 : *          CASSETTE/VIDEO DUMP/READ ROUTINE          *
00130 : * DUMPS SCREEN TO 1024-BYTE CASSETTE RECORD. REAOS   *
00140 : * BACK RECORD AND DISPLAYS ON SCREEN.                 *
00150 : * ENTER AT "WRITE" TO DUMP, "READ" TO READ BACK      *
00160 : *****
00170 :
00180 : SCREEN DUMP
7F00 F3    00185 WRITE DI          ;DISABLE RTC
7F01 AF    00190 XOR      A        ;SELECT CASSETTE 0
7F02 CD1202 00200 CALL    212H     ;TURN ON
7F05 C08702 00210 CALL    287H     ;WRITE LEADER
7F08 21003C 00220 LD      HL,3C00H ;SCREEN START
7F0B E5     00230 WRT010 PUSH HL   ;SAVE PNTR
7F0C 7E     00240 LD      A,(HL)   ;GET VIDEO MEMORY BYTE
7F0D C06402 00250 CALL    264H     ;WRITE TO CASSETTE
7F10 E1     00260 POP      HL       ;GET ADDRESS PNTR
7F11 23     00270 INC      HL       ;BUMP PNTR
7F12 7C     00280 LO      A,H      ;GET MS BYTE
7F13 FE40   00290 CP      40H      ;TEST FOR 4000H
7F15 20F4   00300 JR      NZ,WRT010 ;GO IF NOT ENO
7F17 CDF801 00310 CALL    1F8H     ;TURN OFF CASSETTE
7F1A C9     00320 RET              ;RETURN
00330 ; READ CASSETTE RECORD
7F1B F3     00335 READ  DI         ;DISABLE RTC

```

```

7F1C AF      00340      XOR      A      ;SELECT CASSETTE 0
7F1D CD1202  00350      CALL     £12H    ;TURN ON
7F20 CD9602  00360      CALL     296H    ;READ LEADER
7F23 21003C  00370      LD      HL,3C00H ;SCREEN START
7F26 E5      00380 REA010 PUSH     HL    ;SAVE PNTR
7F27 CD3502  00390      CALL     235H    ;READ BYTE
7F2A E1      00400      POP      HL      ;GET ADDRESS PNTR
7F2B 77      00410      LD      (HL),A  ;STORE BYTE ON SCREEN
7F2C 23      00420      INC      HL      ;BUMP PNTR
7F2D 7C      00430      LD      A,H     ;GET MS BYTE
7F2E FE40    00440      CP      40H     ;TEST FOR 4000H
7F30 20F4    00450      JR      NZ,REA010 ;GO IF NOT END
7F32 18E3    00460      JR      WRT020  ;END ACTION
0000         00470      END
00000 TOTAL ERRORS

```

Figure 10-11. Cassette/Video Program

```

50 CLEAR 100
100 'BASIC ROUTINE WITH EMBEDDED VIDEO/CASSETTE PROGRAM
200 A$=STRING$(52,"")
300 B=VARPTR(A$)
400 B=PEEK(B+2)*256+PEEK(B+1)
500 FOR I=B TO B+51
600 READ A
700 IF I>32767 THEN POKE I-65536,A ELSE POKE I,A
800 NEXT I
900 INPUT "READ(R) OR WRITE(W)":A$
1000 C=B
1100 IF A$="W" GOTO 1300
1200 C=C+27
1300 POKE 16526,C-INT(C/256)*256
1400 POKE 16527,INT(C/256)
1500 INPUT "READY CASSETTE, HIT ENTER":A$
1600 X=USR(0)
1700 GOTO 900
2000 DATA 243,175,205,18,2,205,135,2,33,0,60,229,126,205,100,2
2100 DATA 225,35,124,254,64,32,244,205,248,1,201,243,175,205,18,2
2200 DATA 205,150,2,33,0,60,229,205,53,2,225,119,35,124,254,64
2300 DATA 32,244,24,227

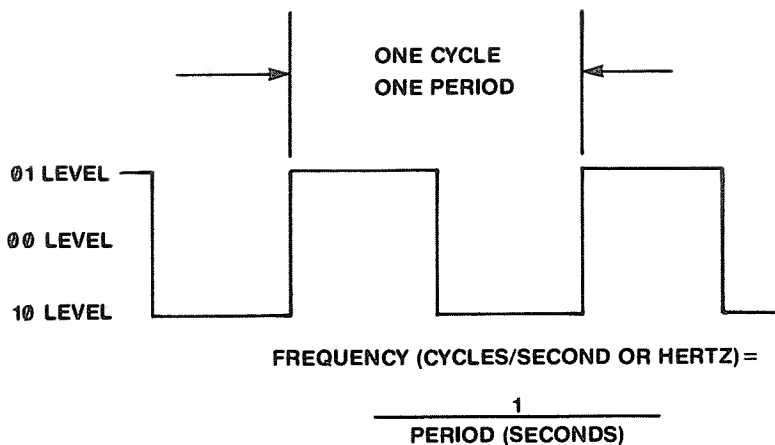
```

Figure 10-12. Cassette/Video BASIC Driver

Cassette Music

As most of you know, the TRS-80 cassette latch is being used to generate musical tones ranging from Morse code to four-voice fugues. The music generated is a constant-level, square-wave tone produced by **toggling** the cassette output bits on and off.

In the simplest case, this involves turning on a positive pulse by 01 and then turning on a negative pulse by 10 bits, as shown in Figure 10-13. The CON subroutine from Chapter 14, for example, toggles the two bits by XORing the current configuration of the bits with 3, which alternates between writing 01 and 10 (see Figure 10-14).



(NOTE: THE TWO PORTIONS OF THE PERIOD
DO NOT HAVE TO BE OF EQUAL
LENGTH)

Figure 10-13. Cassette Square Wave

HL CONTAINS A "DURATION COUNT"

```

04530 :
04540 : ON HERE - GENERATE 500 HERTZ TONE
04550 :
833D 3E01 04560 CON LD A,01 ;ON
833F 2B 04570 DEC HL ;ADJUST FOR "JR C"
8340 01FFFF 04580 LD BC,-1 ;DECREMENT
8343 EE03 04590 ON010 XOR 3 ;TOGGLE
8345 D3FF 04600 OUT (OFFH),A ;OUTPUT TO CASSETTE LATCH
8347 E5 04610 PUSH HL ;SAVE COUNT
8348 210100 04620 LD HL,1 ;FOR 1 MS
834B CDC085 04630 CALL DELAY ;DELAY 1 MS
834E CD2984 04640 CALL INPUT ;GET POSSIBLE CHARACTER
8351 E1 04650 POP HL ;GET COUNT
8352 09 04660 ADD HL,BC ;DECREMENT COUNT
8353 38EE 04670 JR C,ON010 ;GO IF NOT -1
8355 C9 04680 RET ;RETURN

```

Figure 10-14. CON Routine

The delay between the outputs is accomplished by calling the DELAY subroutine, which delays 1 millisecond. As the total **period** of the cycle is 2 milliseconds, a 500 hertz tone is produced for the Morse code dots and dashes. A duration count in HL is decremented down to zero to generate the tone for a given length of time.

In general, what range of tones can be generated by using this method of producing sounds? Producing low tones is no problem because we can delay as long as we wish between toggling the cassette latch output. (Of course, the **fidelity** of the output on either high- or low-frequency tones is another matter.)

The problem here is in producing high-frequency sounds. To do this, we must keep a "tight" loop in the code to toggle the cassette latch between positive and negative pulses. As we're talking about a general case "tone driver," we need some means to vary the period to produce different notes.

About the tightest loop that can be used is:

```

                LO      A,01      ;BIT CONFIGURATION
LOOP1          LO      B,C        ;GET DELAY COUNT
                OUT     (0FFH),A   ;TOGGLE ONE WAY
LOOP2          DJNZ    LOOP2      ;DELAY
                XOR     3          ;INVERT BITS
                JP      LOOP1      ;CONTINUE

```

This takes a delay count in C and puts it into B for each one-half cycle. Either a 01 or 10 is then output to the latch at 0FFH (the motor bit and mode select will be zeroes). The count in B is then decremented down to zero, the bit configuration is inverted, and the process is repeated.

The frequencies that you can produce by this code are easy to figure out. The instructions from LOOP1 through the JP LOOP1 take 2.3, 6.2, 7.3/4.5, 3.9, and 5.6 microseconds, respectively. (We took the T cycle times in the EDTASM manual and multiplied by .56 to get the times in microseconds.) The 7.3/4.5 microsecond time represents the DJNZ time for B<>0 and B=0. The total period for any count in C, is therefore:

PERIOD (microseconds)

$$\begin{aligned}
 &= (2.3 + 6.2 + (CNT - 1) * 7.3 + 4.5 + 3.9 + 5.6) * 2 \\
 &= 43 + 2 * CNT
 \end{aligned}$$

The maximum frequency would be for a minimum period

when CNT=1, which would be a period of 45 microseconds, representing about 22,000 cycles per second (hertz). The minimum frequency would be for a CNT of 0 (256) and would be a period of 555 microseconds for a frequency of 1800 hertz.

There are several problems with this routine. First of all, we need more than just a continuous tone; we need some way of terminating! That means that we must maintain another count for duration. This count is also necessary to produce different durations for musical notes, if we want to implement a full-fledged music program.

Secondly, it appears that we need to keep a larger count in 16 bits for lower frequency notes. The lowest frequency here is 1800 hertz, which is much too high.

There's also another problem, which is not readily obvious. The frequency **resolution** may not be fine enough. The difference in frequencies for a CNT of 200 and 201 is about 10 hertz. This will probably become more pronounced as we add more overhead to the loop.

Hints and Kinks 10-7 Other Tone Parameters

A full-fledged music synthesizer should include some means to control the volume of the output for 'envelope generation' or just plain dynamics. About the best we can do with the cassette output on the TRS-80 is to program two levels - one from the 10 level to the 01 level (low to high) and one from the 10 level to the 00 level (low to reference).

Another parameter that has some interesting effects, however, is the 'duty cycle' of the square wave. We've been working with a 50%-on/50%-off square wave here. However, the harmonic content of square waves will vary with the proportion of on to off time, and you might want to experiment with this parameter as an input for the TONE routine (keeping the period constant).

All this means is that producing musical tones via the cassette port is a compromise (even at best) between range of notes and resolution.

The program in Figure 10-15 is an attempt to produce a wide range of musical notes with good resolution by using two separate routines within the one subroutine, one for high frequencies (HIFREQ) with low overhead and one for low frequencies (LOFREQ) with high overhead.

```

FF00      00100      ORG      OFF00H
00110      ;*****
00120      ;*          TONE GENERATOR
00130      ;* GENERATES HIGH OR LOW FREQUENCY RANGE OF TONES. SYM-
00140      ;* ETRICAL SQUARE WAVE THROUGH CASSETTE OUTPUT.
00150      ;* ENTRY: (HL)=DURATION IN # CYCLES AT FREQUENCY
00160      ;*          (BC)=FREQUENCY COUNT FROM EXTERNAL TABLE
00170      ;*          (A)=0 IF UP TO 300 HERTZ, 1 IF ABOVE 300 HZ
00180      ;*****
0002      00190 DURA DEFS      2          ;DURATION POKED BY BASIC
0002      00200 FREQ DEFS      2          ;FREQ CNT POKED BY BASIC
0001      00210 FLAG DEFS      1          ;FLAG POKED BY BASIC
FF05 F3      00215 TONE DI          ;DISABLE RTC
FF06 2A00FF 00220 LD HL,(OFF00H)      ;GET DURATION
FF09 ED4B02FF 00230 LD BC,(OFF02H)    ;GET FREQ COUNT
FF0D 3A04FF 00240 LD A,(OFF04H)      ;GET FLAG
FF10 B7      00250 OR A          ;TEST HIGH,LOW
FF11 2009      00260 JR NZ,TON010    ;GO IF HIGH
FF13 E5      00270 PUSH HL          ;DURATION TO IX
FF14 DDE1      00280 POP IX
FF16 C5      00290 PUSH BC          ;FREQ CNT TO HL
FF17 E1      00300 POP HL
FF18 CD22FF 00310 CALL LOFREQ      ;GENERATE TONE
FF1B C9      00320 RET
FF1C 0600      00330 TON010 LD B,0    ;CLEAR B OF BC
FF1E CD46FF 00340 CALL HIFREQ      ;GENERATE TONE
FF21 C9      00350 RET
          00360 ; LOW FREQUENCY TONE GENERATOR
FF22 2244FF 00370 LOFREQ LD (FCNT),HL ;STORE FREQ CNT(16)
FF25 11FFFF 00380 LD DE,-1          ;DECREMENT(10)
FF28 2A44FF 00390 LOOP1 LD HL,(FCNT) ;GET FREQ CNT(16)
FF2B 3E01      00400 LD A,1          ;HIGH PULSE(7)
FF2D D3FF 00410 OUT (OFFH),A        ;TURN ON(11)
FF2F 19      00420 LOOP2 ADD HL,DE   ;DECREMENT F CNT(11)
FF30 DA2FFF 00430 JP C,LOOP2        ;GO IF NOT -1(10)
FF33 2A44FF 00440 LD HL,(FCNT)      ;GET FREQ CNT(16)
FF36 3E02      00450 LD A,2          ;LOW PULSE(7)
FF38 D3FF 00460 OUT (OFFH),A        ;TURN OFF(11)
FF3A 19      00470 LOOP3 ADD HL,DE   ;DECREMENT F CNT(11)
FF3B DA3AFF 00480 JP C,LOOP3        ;GO IF NOT -1(10)
FF3E D919      00490 ADD IX,DE       ;DECREMENT DURATION(15)
FF40 DA28FF 00500 JP C,LOOP1        ;GO IF NOT -1(10)
FF43 C9      00510 RET              ;RETURN(10)
0002      00520 FCNT DEFS      2      ;TEMPORARY STORAGE
          00530 ; HIGH FREQUENCY TONE GENERATOR
FF46 11FFFF 00540 HIFREQ LD DE,-1    ;DECREMENT(10)
FF49 41      00550 LOOP4 LD B,C      ;GET FREQ CNT(4)
FF4A 3E01      00560 LD A,1          ;HIGH PULSE(7)
FF4C D3FF 00570 OUT (OFFH),A        ;TURN ON(11)
FF4E 10FE      00580 LOOP5 DJNZ LOOP5 ;GO IF NOT 0(8/13)
FF50 41      00590 LD B,C          ;GET FREQ CNT(4)
FF51 3E02      00600 LD A,2          ;LOW PULSE(7)
FF53 D3FF 00610 OUT (OFFH),A        ;TURN OFF(11)
FF55 10FE      00620 LOOP6 DJNZ LOOP6 ;GO IF NOT 0(8/13)
FF57 19      00630 ADD HL,DE        ;DECREMENT DURATION(11)
FF58 DA49FF 00640 JP C,LOOP4        ;GO IF NOT -1(10)
FF5B C9      00650 RET              ;RETURN(10)
0000      00660 END
00000 TOTAL ERRORS

```

Figure 10-15. Cassette TONE Routine

Two values are used on entry. HL holds the "duration count," while BC holds the "frequency count." The frequency count FD represents the number of iterations through either LOFREQ or HIFREQ to produce a desired frequency. The frequency count FC for any tone less than 300 Hz. can be determined by

$FC = (1/F - 52.45 \text{ microseconds}) / 23.68 \text{ microseconds}$ and for any tone greater than 300 hz. by

$FC = (1/F - 31 \text{ microseconds}) / 14.66 \text{ microseconds}$

These formulas are found by adding up the instruction times in the two routines (T cycle lengths are shown in parentheses).

The duration count DC is simply a count of the number of cycles for any frequency for a given duration in seconds. (For one second duration, this is F cycles).

$$DC = D * F - 1$$

The -1 represents the adjustment for the JP C termination instead of termination on zero.

Typical values for F(frequency), D(duration), FC(frequency count), and DC(duration count) are shown in Figure 10-16.

Frequency (Hz)	Duration (Secs)	Timing Count	Duration Count
100	.25	420.083	24
300	.25	225.261	74
500	.25	134.311	124
700	.25	95.3323	174
900	.25	73.6774	224
1100	.25	59.8971	274
100	.5	420.083	49
300	.5	225.261	149
500	.5	134.311	249
700	.5	95.3323	349
900	.5	73.6774	449
1100	.5	59.8971	549
100	.75	420.083	74
300	.75	225.261	224
500	.75	134.311	374
700	.75	95.3323	524
900	.75	73.6774	674
1100	.75	59.8971	824
100	1	420.083	99
300	1	225.261	299
500	1	134.311	499
700	1	95.3323	699
900	1	73.6774	899
1100	1	59.8971	1099

Figure 10-16. Timing Count Vs. Frequency

Because these counts would be hard to “number crunch” in assembly language, we’ve provided a BASIC driver to interface to the TONE routine (see Figure 10-17). The driver calculates FC and DC from a given frequency and duration and calls TONE with FF00H,1H=DC, FF02H,3H=FC, and FF04H=0 for a tone <300 hz. or 1 for a tone >=300 hz. It would be relatively easy for you to produce musical note frequencies and durations by adding to this driver.

```

100 'TEST DRIVER
200 D=.25
300 FOR F=20 TO 2000 STEP 10
400 GOSUB 10000
500 NEXT F
600 GOTO 600
10000 'TONE DRIVER. ENTER WITH F=FREQUENCY DESIRED, D=DURATION
10010 'IN SECONDS DESIRED.
10020 IF F<300 THEN FC=(1/F-52.43E-6)/23.68E-6 ELSE
FC=(1/F-31E-6)/14.66E-6
10030 OC=D*F-1
10040 POKE &HFF00,DC-INT(OC/256)*256
10050 POKE &HFF01,INT(OC/256)
10060 POKE &HFF02,FC-INT(FC/256)*256
10070 POKE &HFF03,INT(FC/256)
10080 IF F<300 THEN POKE &HFF04,0 ELSE POKE &HFF04,1
10090 OEFUSRO=&HFF05
10100 X=USRO(0)
10110 RETURN

```

**Figure 10-17. BASIC TONE
Driver**

Chapter Eleven

Disk I/O in Assembly Language

We're going to look at another type of I/O device in this chapter — the floppy disk drive of the TRS-80. We'll first examine some of the physical and electrical characteristics of the disk, look at TRSDOS file structure, and then look at communication with disk data via assembly-language calls to TRSDOS.

Diskette and Disk Characteristics

A diskette is a circular piece of mylar coated with a ferro-magnetic material. As it comes from the manufacturer, it's unmagnetized with **no data of any type** on it. There are no inherent **tracks** or **sectors** permanently embedded in the magnetic medium.

The manufacturer usually **certifies** the diskette. This certification process involves writing and reading back data at high **bit densities** to verify that there aren't any gaps or flaws in the magnetic material that would cause loss of data.

There are two basic diskette formats, the **hard-sectored** and **soft-sectored**. The TRS-80 uses a soft-sectored type of diskette format. The hard sectored diskette has ten sector index holes, while the soft-sectored diskette has one sector index hole, as shown in Figure 11-1.

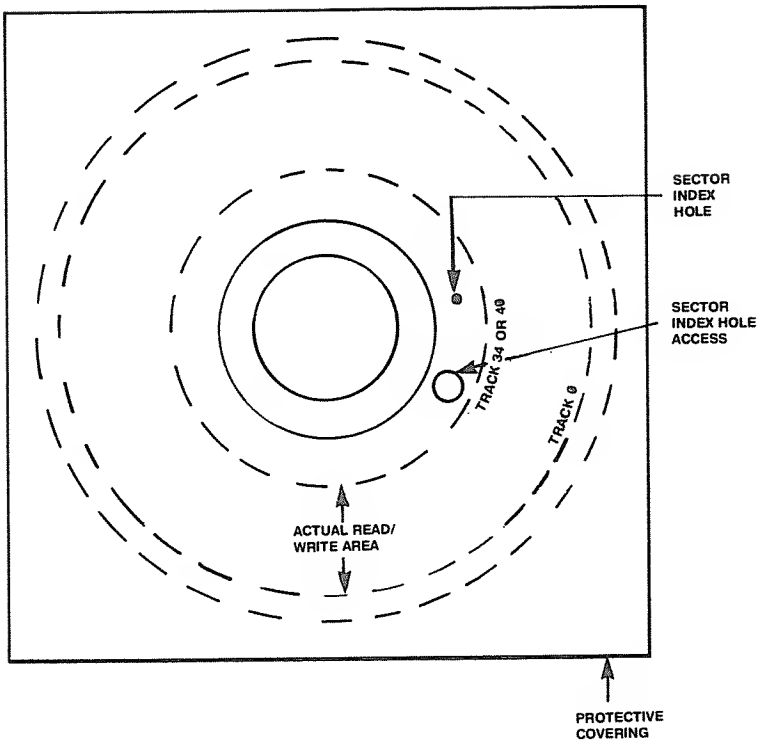


Figure 11-1. Diskette Structure

The purpose of the sector index hole is to act as a reference for the start of **sector 0**, the first sector of the ten sectors per **track** for the Model I or 18 sectors per track for the Model III. Each sector is filled with 256 bytes of data, making a total track's worth of data 2560 bytes (4608 bytes for the Model III). Each diskette is normally divided into 35 tracks for the Model I or 40 tracks for the Model II, numbered 0 through 34 (39). The entire diskette can therefore hold $2560 \times 35 = 89600$ bytes of data (184320 bytes for the Model III).

In addition to the data bytes in each sector, there's **header** and **trailer** data in the sector. This data contains the track #, sector number, and checksum for the sector. In addition, there is "filler" data preceding and trailing the ten sectors worth of data, as shown in Figure 11-2.

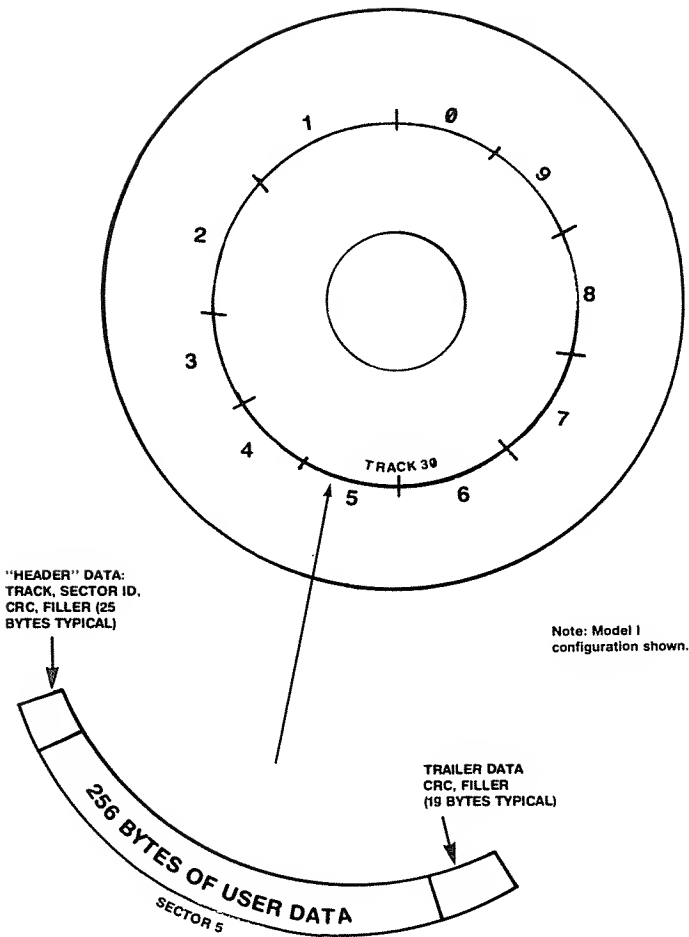


Figure 11-2. Disk Header/Trailer

All of this data is put on the disk by the **formatting program**. The formatting program is a simple program that you use to **format** the header, trailer, and filler data on the diskette in preparation for storing actual data in the data portion of each sector. You might visualize the action of the formatting program as putting a "skeleton" of the sector structure on disk. The formatting program simply stores 256 bytes of dummy data for the 256-byte data area in each sector.

Hints and Kinks 11-1

How is Formatting Done?

You don't really want to format your own disks, do you? All right, here goes

Formatting simply involves positioning the disk head to each of the 35 (Model I) or 40 (Model 11I) tracks and then issuing a "write track" command to the disk controller chip. The write track essentially tells the controller "here comes the data." The data consists of ten sector segments and filler. The sector segments contain special characters that cause data address "marks," 1D address marks, and CRC check bytes to be written. The actual sequence is FE (1D address mark action), track #, sector #, F7 (CRC action), filler, data address mark, dummy user data (256 bytes of E5), F7 (CRC action), and filler, repeated 10 times.

The formatter program writes this data out one byte at a time until the entire track has been written.

Disk Drives

In order to locate a track, each 35- or 40-track disk drive steps or positions a read/write head one discrete increment for each track, as shown in Figure 11-3. There are no other track references that the drive can read to find the proper track; it has to step the head precisely for each increment to a new track.

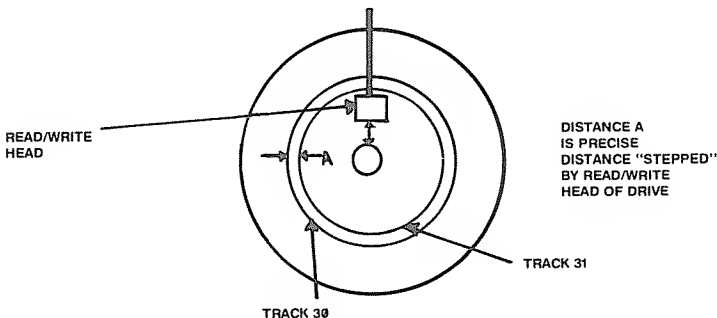


Figure 11-3. Disk Drive Operation

To perform this task, each drive has a stepper motor or other positioning scheme that positions the head precisely. There are three basic movements that each drive can perform: it can **restore** the head to a position over track zero, it can step inward one track; or it can step outward one track. The software controlling the drive usually does a restore operation initially, and then keeps track of where the head is as it steps in and out over the various tracks. At any time, however, it can always easily find track zero again by performing a restore.

The disk drive motor turns at about 300 revolutions per minute (5 revolutions per second). This means that data is passing under the read/write head of the drive at about 2560×5 bytes per second (12,800 bytes per second) for the Model I and about 4608×5 bytes per second (23,040 bytes per second) for the Model III. Since the data is recorded **serially** along the track, this amounts to a string of 102,400 (184,320) bits recorded along each track. (In these figures we're ignoring the header, trailer, and filler data, which actually increases the **data rate** by 20% or so.)

—Hints and Kinks 11-2—

Disk I/O Read/Write Timing

Considering that the data rate is about 12,800 bytes per second, that means a byte every 78 microseconds for the Model I. Since the data is being written out a byte at a time in a software timing loop, the loop itself must be fairly 'tight' to ensure that it can get back with the next byte in time. If you miss that 78 microsecond 'window,' you end up with the dreaded lost data condition. The ball game is lost, at least for that sector. A time of 78 microseconds represents, say, 15 instructions, so some efficiency in coding is called for here, especially if real-time-clock interrupt processing is occurring every 25 milliseconds or so! (RTC processing adds more instructions.) For the Model III, the timing constraints are even 'tighter' — about a byte every 43 microseconds.

The disk drive is really a very **dumb** device. The signals passing from the disk drive to the expansion interface are shown in Figure 11-4. There aren't very many. There are two lines for WRITE DATA and READ DATA. Since the disk drive reads and writes data serially along the track, data is passed a bit at a time in one direction or the other. There's also a line that passes a STEP command to the drive; associated with the STEP line is a DIRECTION line which determines whether the track step will be in or out.

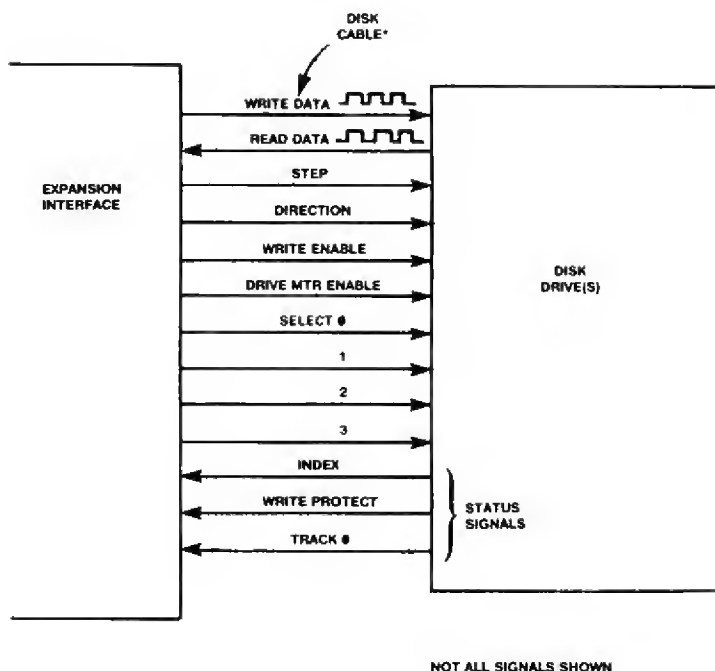


Figure 11-4. Disk Signals

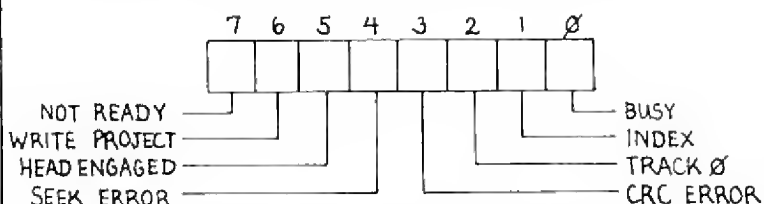
WRITE ENABLE must be in force to allow writing data to the disk. You use DRIVE MTR ENABLE to turn on the disk motor for a read or write and SELECT to select the drive and enable the operation.

Signals coming back include the READ DATA line, and the **status** signals INDEX (true when index hole passes under a sensor), TRACK ZERO (true when the head is over track zero), and WRITE PROTECT (true when the write protect diskette notch is covered).

Hints and Kinks 11-3

Status Signals

Like other I/O devices, the disk and disk controller send back status so that the program knows about disk conditions and about the busy/read state for reads and writes. The status byte for a positioning command such as a restore command on the Model I would be



As a fun project, load the following BASIC program to test 'sector 0' (index hole) status for a Model I disk drive:

```

100 POKE 14304,1
200 A=PEEK(14316)
300 IF (A AND 2)=2 PRINT "SECTOR 0"
400 PRINT A
500 GOTO 200

```

The Disk Controller

As the disk drive is such a dumb device, intelligence for disk drive operations must be incorporated elsewhere. Much of the intelligence is in the **disk controller chip** in the expansion interface of the TRS-80 Model I or the disk controller logic of the Model III.

The disk controller chip is a small microprocessor in itself. It handles all of the lower-level disk operations such as

- Converting 8-bit bytes into eight serial bits for writes

- In doing all of these operations, the disk controller relieves the software from controlling the timing and other functions as in the cassette drivers.

The expansion interface of the Model I contains the disk controller chip and some additional circuitry for disk drive address decoding. A block diagram of this is shown in Figure 11-5. There are two general sets of addresses associated with the disk. Both are **memory-mapped addresses** that are 16-bit values. (Memory reference instructions are used in place of INs and OUTs.)

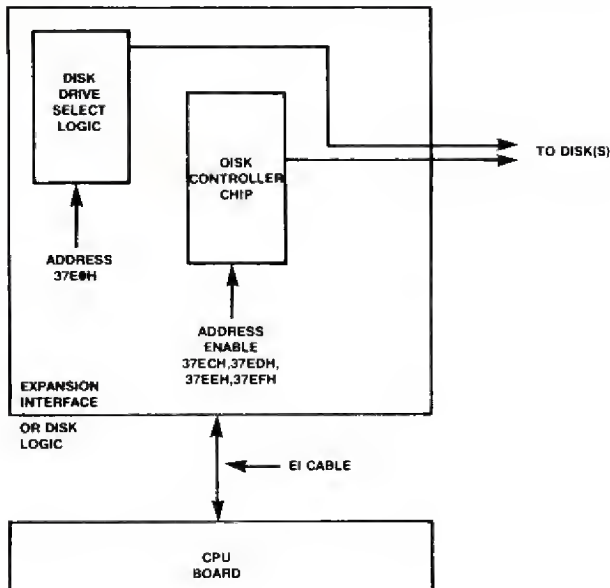


Figure 11-5. Disk Logic

One of the four possible disk drives on a TRS-80 Model I system is “selected” by loading a register with a 1, 2, 4, or 8 and performing a write to memory location 37E0H. This operation essentially turns the drive motor on for about three seconds and **enables** other operations to take place. A typical select of drive 0 would be

```
LD A,1          ;FOR DRIVE 1
LD (37E0H),A     ;SELECT DRIVE OUT 64H 244
                64H,A
```

The remaining addresses are 37ECH, 37EDH, 37EEH, and 37EFH.

These addresses are all associated with registers in the disk controller chip. The general actions for each address for a read or write are

ADDRESS	READ ACTION	WRITE ACTION
37ECH	Read status	Write command
37EDH	Read track	Write track
37EEH	Read sector	Write sector
37EFH	Read data	Write data

All communications from TRSDOS and other software are done with the disk controller chip by issuing a series of one-byte commands and by transferring one byte of data between a CPU register and the disk controller chip.

For the Model III, the procedure is very similar, except that the “memory-mapped” addresses have been changed to I/O ports of 0F0H, 0F1H, 0F2H, and 0F3H.

Hints and Kinks 11-4
Some Disk Controller Commands

Commands available to the program for the disk controller include eleven separate one-byte instructions. These include five commands to position the head: Restore puts the head over track 0; Seek finds a given track; Step, Step-In, and Step-Out all move the head either in or out one track.

Read and Write are used to start the process of writing one sector's worth of data. Prior to the read or write, the head must have been positioned over the proper track, and a sector register must have been loaded with the sector address. The read or write is followed by the transfer of 256 bytes in a software loop that looks for a status bit indicating that the controller can accept the next data byte.

Read Address reads the track/sector address (not often used). Read Track reads the entire formatted track; Write Track writes (formats) a track. Force Interrupt causes a software interrupt.

Unfortunately, a large number of commands can be issued to the controller chip with complicated actions and responses — far too many to adequately cover in one chapter. So we'll give you a typical sequence of operations, a flavor of how things are done in **disk driver** programs that make up the lowest level of TRSDOS and other programs that read and write data to the disk. Figure 11-6 shows the disk bootstrap program in Level II ROM. It first tests to see whether a disk drive is connected to the system, restores the disk head to track 0, and then reads in the 256 data bytes of sector 0, track 0. This sector contains a **bootstrap program (BOOT/SYS)** that reads in the remainder of TRSDOS.

0696		00050	ORG	696H	
0696	3AEC37	00100	LO	A,(37ECH)	;GET DISK STATUS
0699	3C	00110	INC	A	
069A	FE02	00120	CP	02	
069C	0A7500	00130	JP	C,75H	;GO IF NO DISK
069F	3E01	00140	LO	A,1	;FOR ORIVE 1
06A1	32E137	00150	LO	(37E1H),A	;SELECT ORIVE 1
06A4	21EC37	00160	LO	HL,37ECH	;ADDRESS COMMAND
06A7	11EF37	00170	LO	OE,37EFH	;DATA REGISTER A00R
06AA	3603	00180	LO	(HL),3	;RESTORE COMMAND
06AC	010000	00190	LO	BC,0000	;DELAY 64K COUNTS
06AF	C06000	00200	CALL	0060H	
06B2	CB46	00210	LOOP1 BIT	0,(HL)	;TEST BUSY
06B4	20FC	00220	JR	NZ,LOOP1	;GO IF STILL BUSY
06B6	AF	00230	XOR	A	;ZERO A
06B7	32EE37	00240	LO	(37EEH),A	;0 TO SECTOR REG
06BA	010042	00250	LO	BC,4200H	;MEMORY ADDRESS
06B0	3E8C	00260	LO	A,8CH	;READ COMMAND
06BF	77	00270	LO	(HL),A	;READ SECTOR 0
06C0	CB4E	00280	LOOP2 BIT	1,(HL)	;TEST DRQ
06C2	28FC	00290	JR	Z,LOOP2	;GO IF NO DATA AVAIL
06C4	1A	00300	LO	A,(OE)	;GET NEXT DATA BYTE
06C5	02	00310	LO	(BC),A	;TRANSFER DATA
06C6	0C	00320	INC	C	;BUMP BUFFER PTR
06C7	20F7	00330	JR	NZ,LOOP2	;GO IF NOT 256 BYTES
06C9	C30042	00340	JP	4200H	;TRANSFER TO LOADER
0000		00350	ENO		
00000	TOTAL ERRORS				

Note: Model I Bootstrap
Shown.

LOOP TO READ 256
BYTES OF TRACK 0,
SECTOR 0 INTO 4200H
AREA

Figure 11-6. Disk Bootstrap

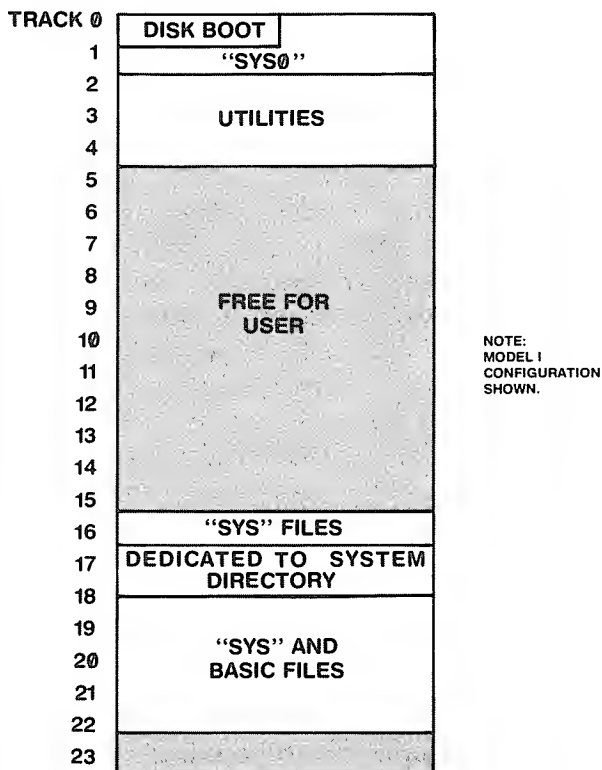
TRSDOS Disk Organization

Actually there's no need to write your own disk driver routines for the TRS-80. There're a number of TRSDOS assembly-language routines that will handle almost all disk operations you'd want to perform. These routines incorporate not only code to perform rudimentary disk operations such as reading a sector and restoring the head to track 0, but code to perform **disk file manage** operations on the TRS-80. Disk file manage refers to operations to locate, read, and write disk files. We'll discuss all of these routines in detail, but first let's look at TRSDOS disk organization so that we may better understand what is involved.

TRSDOS disk files are made up of from 1 to 32 **granules**. A granule is five (Model I) or three (Model III) sectors and is the minimum amount of space that TRSDOS allocates when establishing a new file or adding to an existing file.

TRSDOS uses **dynamic disc allocation** that allocates only enough (or slightly more) disk space to hold the current number of **granules** in a file. Deleted files cause the disk space used in the file to be released to the **pool** of disk granules (there is a **granule allocation table** or GAT that is essentially a directory of which granules are in use and which are in the pool of unused granules).

As you know from reading your TRSDOS manual, there may be up to 48 separate user files on each diskette. A file is simply a collection of **records** which are sets of any type of data in some organized fashion. In practice, a file may be spread over **non-contiguous** areas of the disk, but you can use the file manage functions of TRSDOS to retrieve all records associated with a file by reference to the **file directory** on disk track 17. A typical disk map for the Model I is shown in Figure 11-7.



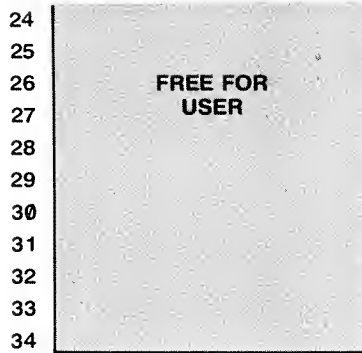


Figure 11-7. Typical Disk Map

A **physical record** on disk is equivalent to a disk sector. At the disk driver level, all disk files are read one sector at a time, bringing in 256 bytes of data. However, within a physical record there may be one or more **logical records**.

A logical record is the actual record used in data storage and retrieval by the program. It may be 1 to 255 bytes in length. Logical records are **blocked** into physical records (sectors) by the TRSDOS file manage routines. For example, if you used 64-byte logical records to hold names and phone numbers, four 64-byte records would be packed into the sector physical record. One of the main tasks of the TRSDOS file manage routines is to access the next logical record from the proper physical record.

TRSDOS works with a **buffer** in memory. A buffer is an area of memory set aside to hold the 256 bytes of a physical record sector. When we're working in Disk BASIC, these buffer areas are preassigned to fixed locations in the TRSDOS area; however, we can use our own areas, when we're using the TRSDOS routines, as we shall see.

Device Control Blocks

Device Control Blocks are "working storage" areas of memory dedicated to variables connected with a particular I/O driver. Level II BASIC uses several DCBs connected with keyboard, video display, and line printer operations; TRSDOS

uses additional DCBs for disk operations. In both cases the location of the DCB is fixed. When we utilize the TRSDOS disk file manage calls, however, we can place the DCBs anywhere in memory we desire and pass the location of the DCB as a parameter.

We may have as many DCBs as we require. For example, if we are merging two files on disk into a third file, we could have three DCBs, one for the "old master file," one for the "transaction file," and one for the "new master file," as shown in Figure 11-8.

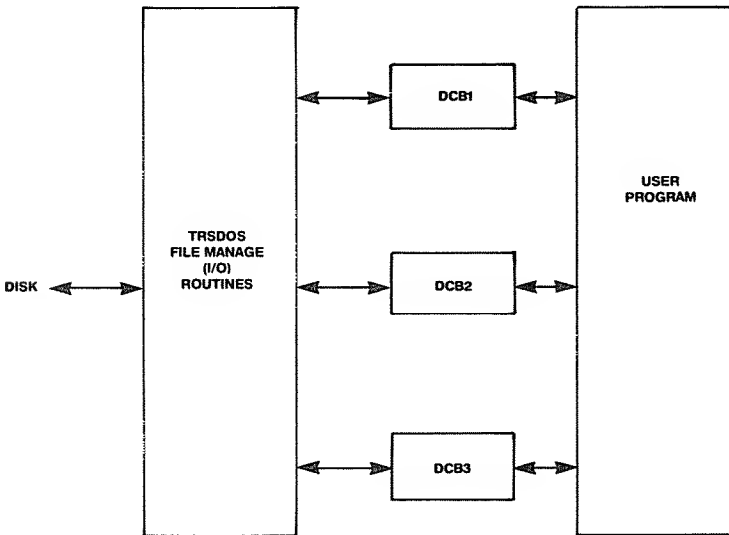


Figure 11-8. DCB Use

Each DCB has two lives. Before the file operations are started (OPENed) and after file operations are terminated (CLOSEd), the 32-byte (Model I) or 30-byte (Model III) DCB contains the file name in standard TRSDOS format, a carriage return, and blanks, as shown in Figure 11-9. During file operations, the TRSDOS file manage routines and the user communicate (pass parameters) by using variables in the DCB as shown.

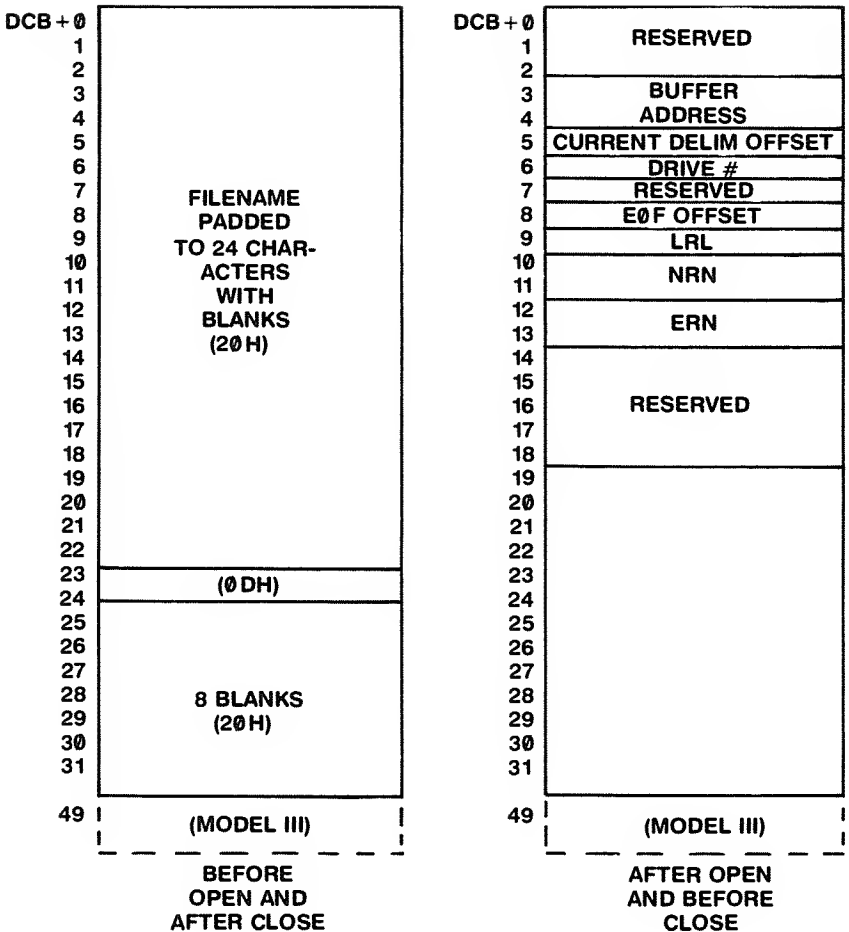


Figure 11-9. DCB Format

During the file operations, the DCB bytes are used to hold variables. The variables are put into the DCB by the TRSDOS routines but may be examined by your assembly-language code. Some of the variables are initialized from user-supplied data, such as the buffer address.

Bytes +3 and 4 hold the buffer address for disk reads and writes. The buffer may be anywhere in memory that's compatible with the configuration; without Disk BASIC, this

means anywhere above 6FFFH (Model I) or 6000H (Model III). If Disk BASIC is being used, follow the same rules about assembly-language memory areas as you normally would (see Chapter 4).

Byte + 5 holds a buffer displacement (or offset) of 1 to 255 to the end of the current record. In other words, it's an index to the last byte of the current record. If, for example, you were processing the second 64-byte logical record of a sector, this byte would contain 127. During normal processing, you would not employ this variable.

Byte + 6 holds the file drive number of the disk drive being used in the operation. Valid numbers are 0 through 3. This constant is stored by TRSDOS from the file name of the DCB.

Byte + 8 holds the buffer displacement (offset) of the last delimiter of the last physical record. In other words, when the last sector has been read in, this byte contains an index of 1 to 255 that points to the last byte of the file. Don't use this variable during normal processing.

Byte + 9 holds the LRL, or logical record length. This variable may be 1 to 255 and is put in by TRSDOS from user-supplied data. If the LRL is 0, TRSDOS will assume the user is **blocking** his own records and will not find logical records for him (more about that later). This variable is constant for the entire file.

Bytes + 10 and + 11 hold the NRN or next record number in standard 16-bit format (ls byte followed by ms byte). This variable is set to 000H by TRSDOS after the file has been found initially (OPENed) or created (INITialized). It is incremented by one by TRSDOS after each read for **sequential files**, or you may change it for **random files**.

Bytes + 12 and + 13 contain the ending record number of the current file in standard 16-bit format. This represents the last logical record number of the file.

Byte + 0, + 1, + 2, + 3, + 14, + 15, + 16, + 17, and + 18 are "reserved," which is a polite way of saying, "Don't tread on me!" They should not be altered by the user.

TRSDOS I/O Calls

Now that we know how the DCB is set up, let's look at the sequence and setup for TRSDOS I/O calls. There is an excellent write-up of the calls in the *TRSDOS/Disk BASIC Reference Manual*, so we won't repeat the formats here. There are eight calls for the Model I and seventeen calls for the Model III. We'll consider the "subset" of eight Model I calls here:

Call	Action
INIT	Initialize a new file
OPEN	"Open" an existing file
POSN	Positions a file to read or write a random record
READ	Reads one logical record from disk or buffer
WRITE	Writes one logical record into disk or buffer
VERF	Verifies a physical record
CLOSE	"Closes" an opened file
KILL	Closes a file and deletes it from directory

In general, all of these calls are made by CALLING a TRSDOS routine in the 44XXH area. In all calls, the DE register holds a pointer to the first byte of the DCB. HL and B or BC may also hold a parameter depending upon the call type. For all calls, after the call is made, a successful action returns with the Z flag set. If the Z flag is not set on return, an error has occurred and the A register contains an error code. Error codes for TRSDOS I/O calls, along with probable causes, are shown in the *TRSDOS/Disk BASIC Reference Manual*.

Reading an Existing I/O File

The normal sequence to read an existing file is this:

1. Put the file name in standard format into the DCB.
2. Make an OPEN call. This causes TRSDOS to search the directory and find the file.
3. If the file is a sequential file, perform a series of READS of the next logical record until the last record (ERN or ending record number) has been processed. If the file is a random file, perform the READs after a POSN call for

each READ. The POSN uses the desired NRN (next record number) in the DCB to find the required logical record.

4. CLOSE the file.

To see how this works in a simple case, enter the BASIC program shown in Figure 11-10. Save the program on disk by performing an ASCII save:

SAVE "TEST",A

```

100 'THIS IS LINE 1.....
200 'THIS IS LINE 2.....
300 'THIS IS LINE 3.....
400 'THIS IS LINE 4.....
500 'THIS IS LINE 5.....
600 'THIS IS LINE 6.....

```

**Figure 11-10. BASIC ASCII
Test File**

This will write the file in ASCII format so that we can use the program in Figure 11-11 to read it in a record at a time and list it on the screen.

```

8000          00100      ORG      8000H
3C00          00110  BUFFER EQU      3C00H
              00120      ; SAMPLE PROGRAM TO READ EXISTING FILE
              00130      ;
              00140      ; FIRST OPEN THE FILE
8000 21003C    00150  READF  LD      HL,BUFFER      ;BUFFER LOCATION IN HL
8003 113B80    00160          LD      DE,DCB1      ;DCB LOCATION
8006 0600      00170          LD      B,0         ;READ ONE SECTOR
8008 CD2444    00180          CALL   4424H        ;MAKE OPEN CALL
800B 2808      00190          JR      Z,REA010     ;GO IF OK
800D F680      00200  REA005 OR      80H          ;SETUP FOR ERROR MSG
800F CD0944    00210          CALL   4409H        ;DISPLAY ERROR MESSAGE
8012 CD2D40    00220          CALL   402DH        ;REBOOT
              00230      ; NOW READ AND DISPLAY
8015 113B80    00240  REA010 LD      DE,DCB1      ;DCB LOCATION
8018 CD3644    00250          CALL   4436H        ;READ RECORD
801E 20F0      00260          JR      NZ,REA005   ;GO IF ERROR
801D 2A3E80    00270          LD      HL,(DCB1+3)  ;GET BUFFER ADDRESS
8020 110001    00280          LD      DE,256      ;INCREMENT
8023 19        00290          ADD     HL,DE       ;BUMP TO NEXT SCREEN SECTION
8024 223E80    00300          LD      (DCB1+3),HL ;STORE FOR NEXT READ
8027 DD213B80  00310          LD      IX,DCB1     ;DCB ADDRESS
802B DD7E0A    00320          LD      A,(IX+10)    ;GET NRN
802E DDEE0C    00330          CP      (IX+12)     ;COMPARE TO LRN
8031 20E2      00340          JR      NZ,REA010   ;GO IF ERROR
8033 113B80    00350          LD      DE,DCB1     ;DCB LOCATION
8036 CD2844    00360          CALL   4428H        ;CLOSE FILE
8039 18FE      00370  RFA020 JR      REA020      ;JUMP HERE ON END
803B 54        00380  DCB1  DEFM  'TEST'

      45 53 54 20 20 20 20 20 20
      20 20 20 20 20 20 20 20 20
      20 20 20 20 20 20 20
8052 20        00390          DEFM  ' '
      20 20 20 20 20 20 20 20
0000          00400          END
00000 TOTAL ERRORS

```

**Figure 11-11. Read Test File
Program**

The program in Figure 11-11 is a simple read of an existing disk file. We did some things in it that are a little dangerous (like incrementing 256 bytes each time through the loop for the screen address and modifying the buffer address in the DCB), so try it only with the TEST file from Figure 11-10.

The DCB for the read is DCB1. Before we OPEN the file, the DCB contains TEST padded out to 23 characters with blanks, a carriage return, and eight terminating blanks. This corresponds to the DCB format required by the TRSDOS calls. We OPENED the TEST file by CALLING 4424H with the buffer address in HL, the DCB address in DE, and the logical record length in B. The logical record length in this case is 0, signifying that a logical record corresponds to a physical record, or sector.

If an OPEN error had occurred, Z would not be set on return, and we would have gone to the special "display TRSDOS error message" routine at 4409H and then rebooted.

Hints and Kinks 11-5 Error Code Routine

The error code routine is another TRSDOS call that converts the error code returned in the A register to a description message. The message is then displayed. It makes sense to use it, rather than printing out a nebulous ERROR 1786, SUB A-B DURING HIGHEST HIGH TIDE.

After a successful open, the DCB contains variable data placed in it by the TRSDOS OPEN routine. The DCB appears as shown in Figure 11-12. Note that at this time TRSDOS knows the length of the TEST file and puts a 0002H in DCB+12,+13 (ERN or end record number). It has also initialized the NRN (next record number) to 0000H in preparation for a READ or WRITE.

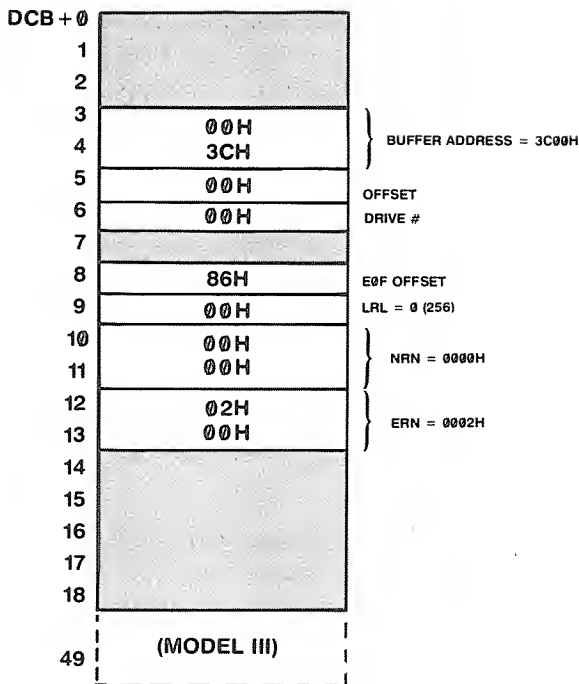
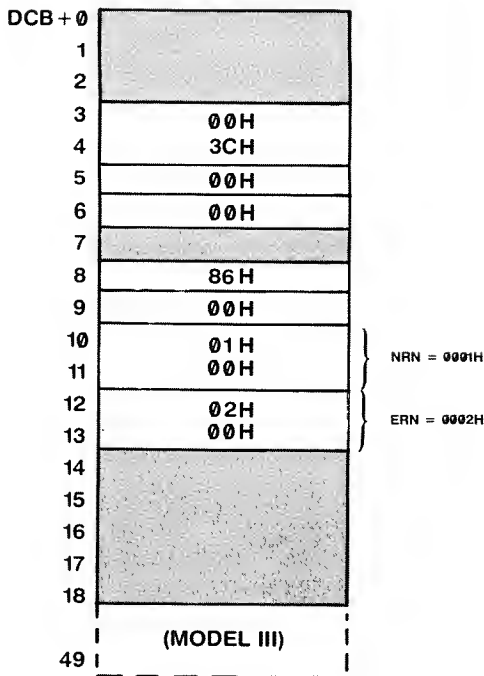


Figure 11-12. DCB After OPEN

The next set of code makes a call to READ a record. If the logical record length were not 0 (256 bytes), we would specify a "user record" area in HL. The READ call would then transfer the next logical record from the buffer into the user record area. This might involve a new disk read of the next sector, or would simply involve transferring the next logical record from the buffer.

Since we're working with logical records equal to physical records (sectors), however, we did not pass anything in HL.

After a successful READ, the DCB appears as shown in Figure 11-13. The NRN has been incremented to 1, and the data from the sector appears on the screen (buffer). The buffer location is now picked up from DCB+3,+4 and incremented by 256 to point to the next screen location. (Don't try this for more than four sectors!)



**Figure 11-13. DCB After
READ**

Now we come to an important portion of code. The current record number is picked up from DCB+10 and compared to the ending record number in DCB+12. If they are not equal, another READ is done. If they are equal, you perform a CLOSE file by CALLing TRSDOS CLOSE at 4428H with DE containing the DCB address. For the TEST file, we make two passes through the read portion of the code and then CLOSE the file and loop at REA020.

This is in essence how a read of any existing sequential file can be accomplished. Many Radio Shack disk files have logical record lengths of 256 bytes (look at any DIR display), so the procedure for processing any existing BASIC files would be very similar to the above code.

Creating a New File and Reading It Back

Now that we've gotten our feet wet with disk I/O, let's try a plunge into deep water with a write of a new file and a following read. We'll use logical record lengths other than 256. The program shown in Figure 11-14 takes the contents of the screen and opens a new file called SCREEN. SCREEN has logical record length equal to the size of a screen line (64 bytes). There'll be 16 logical records in 4 physical records for the SCREEN file. After the file is created it can be read back and displayed by the second part of the program.

The program is divided into three parts — initializing a new file and writing it, clearing the screen, and reading the file back in.

Initializing

The first part calls INIT to create a new file. A 256-byte buffer is specified in HL, and the usual DCB is specified in DE. The DCB in this case has the name "SCREEN". **A logical record length of 64 is specified in B.** The code at WRT005 is an error routine to display any disk error on the screen.

After a successful INIT, the character string in the DCB is replaced with the date shown in Figure 11-15. Note that the NRC (next record number) is initialized to 0000H for the INIT, just as it was for the OPEN.

```

8000          00100          ORG      8000H
00110      ; SAMPLE PROGRAM TO WRITE SCREEN TO OISC AND READ IT BACK
00120      ; WRITE SCREEN

8000 217280  00130  WRTREA  LO      HL,BUFFER      ;BUFFER LOCATION
8003 117281  00140          LO      OE,WOCB        ;OCB LOCATION
8006 0640    00150          LO      B,64           ;64 BYTES PER LOG RECORD
8008 C02044  00160          CALL   4420H          ;CREATE A NEW FILE
800B 2808    00170          JR      Z,WRT010        ;GO IF OK
8000 F680    00180  WRT005  OR      80H           ;SETUP FOR ERROR MSG
800F C00944  00190          CALL   4409H          ;WRITE ERROR MSG
8012 C32040  00200          JP      4020H          ;REBOOT
8015 21C03B  00210  WRT01C  LO      HL,3C00H-64    ;SCREEN START-256
8018 E5      00220          PUSH   HL             ;SAVE ADDRESS
8019 E1      00230  WRT020  POP      HL            ;GET LINE #
801A 114000  00240          LO      OE,64          ;INCREMENT
8010 19      00250          ADO     HL,OE          ;POINT TO NEXT LINE
801E 7C      00260          LO      A,H           ;GET MS BYTE
801F FE40    00270          CP      40H           ;TEST FOR LAST LINE
8021 280B    00280          JR      Z,WRT030        ;GO IF OONE
8023 E5      00290          PUSH   HL             ;SAVE UREC ADDRESS
8024 117281  00300          LO      OE,WOCB        ;WRITE OCB LOCATION
8027 C03944  00310          CALL   4439H          ;WRITE UREC
802A 20E1    00320          JR      NZ,WRT005      ;GO IF ERROR
802C 18EB    00330          JR      WRT020         ;CONTINUE
802E 117281  00340  WRT030  LO      OE,WOCB        ;OCB LOCATION
8031 C02844  00350          CALL   4428H          ;CLOSE FILE
8034 2007    00360          JR      NZ,WRT005      ;GO IF ERROR
00370      ; CLEAR SCREEN
8036 21003C  00380          LO      HL,3C00H        ;SCREEN START
8039 3E20    00390  WRT040  LO      A,' '          ;BLANK
803B 77      00400          LO      (HL),A         ;STORE BLANK
803C 23      00410          INC     HL             ;BUMP PTR
8030 7C      00420          LO      A,H           ;GET MS BYTE
803E FE40    00430          CP      40H           ;TEST FOR ENO
8040 20F7    00440          JR      NZ,WRT040      ;GO IF NOT ENO
00450      ; NOW READ BACK FILE
8042 217280  00460          LO      HL,BUFFER      ;BUFFER LOCATION
8045 117281  00470          LO      OE,WOCB        ;OCB LOCATION
8048 0640    00480          LO      B,64           ;64 BYTES PER LOG RECORD
804A C02444  00490          CALL   4424H          ;OPEN SCREEN FILE
804D 20BE    00500          JR      NZ,WRT005      ;GO IF ERROR
804F 21C03B  00510  WRT050  LO      HL,3C00H-64    ;SCREEN START-256
8052 E5      00520          PUSH   HL             ;SAVE ADDRESS
8053 E1      00530  WRT060  POP      HL            ;GET LINE #
8054 114000  00540          LO      OE,64          ;INCREMENT
8057 19      00550          ADO     HL,OE          ;POINT TO NEXT LINE
8058 7C      00560          LO      A,H           ;GET MS BYTE
8059 FE40    00570          CP      40H           ;TEST FOR LAST LINE
805B 280B    00580          JR      Z,WRT070        ;GO IF OONE
8050 E5      00590          PUSH   HL             ;SAVE UREC ADDRESS
805E 117281  00600          LO      OE,WOCB        ;OCB LOCATION
8061 C03644  00610          CALL   4436H          ;READ UREC
8064 20A7    00620          JR      NZ,WRT005      ;GO IF ERROR
8066 18EB    00630          JR      WRT060         ;CONTINUE
8068 117281  00640  WRT070  LO      OE,WOCB        ;OCB ADDRESS
806B C02844  00650          CALL   4428H          ;CLOSE FILE
806E 2090    00660          JR      NZ,WRT005      ;GO IF ERROR
8070 18FE    00670  WRT080  JR      WRT080        ;LOOP HERE
0100          00680  BUFFER  OEFS      256
8172 53      00690  WOCB   OEFM      'SCREEN
      43 52 45 45 4E 20 20 20
      20 20 20 20 20 20 20 20
      20 20 20 20 20 20
8189 00      00700          OEFB      00H
818A 20      00710          OEFM      '
      20 20 20 20 20 20 20 20
0000          00720          ENO
00000 TOTAL ERRORS

```

Figure 11-14. SCREEN Program

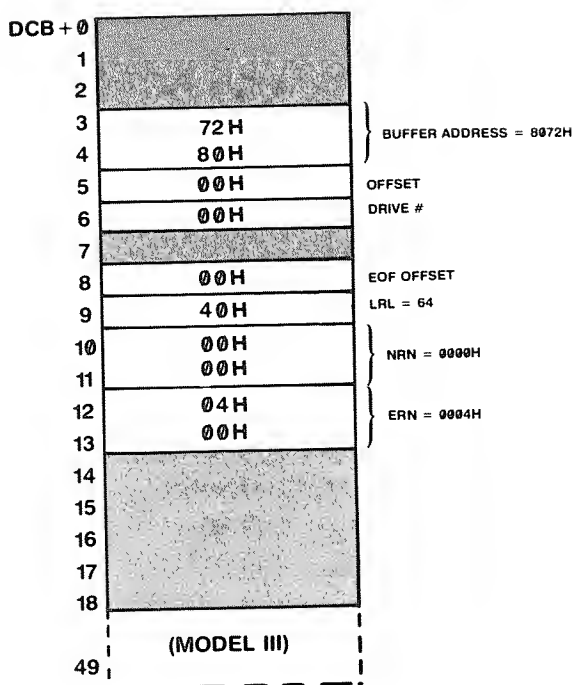


Figure 11-15. DCB After INIT

Writing

Now a series of disk writes is done. Each write specifies a UREC (user record area) in HL. This would normally be RAM memory, but in this case we're using the video screen as a user buffer. The user buffer must be the same length as the logical record length or greater. In fact each new call to WRITE uses the next screen line as the UREC. Each write causes 64 bytes to be moved from a screen line into the buffer area. After each fourth move, a disk write of the physical record (sector) is automatically performed. Typical DCB contents are shown in Figure 11-16.

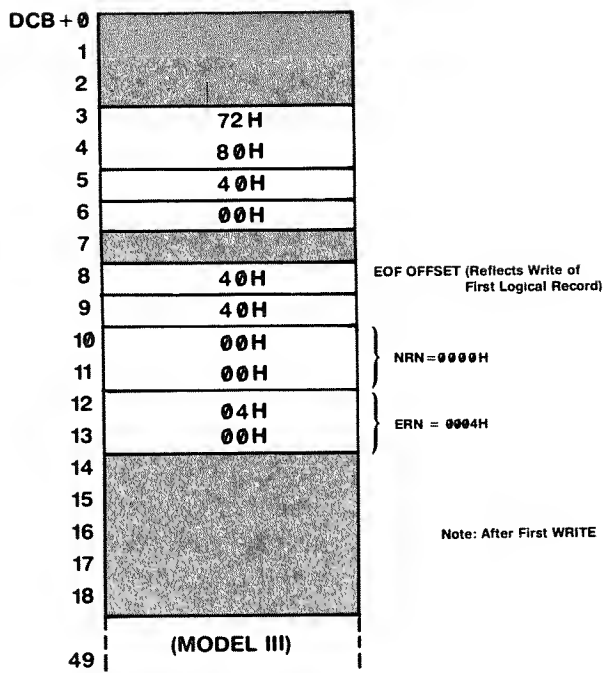


Figure 11-16. DCB After WRITE

Clearing

After 16 writes of the 16 screen lines, a call is made to CLOSE to close the SCREEN file. The CLOSE is especially important on a write as there may be logical records in the buffer that have not yet been written to disk. The CLOSE in this case **flushes** the buffer.

The screen is cleared with the next bit of code.

Now an OPEN is done with the same DCB used as in the INIT and WRITE. Note that the CLOSE **restored** the character data in the DCB, and that we can perform the OPEN without having to reinitialize the DCB with the file name. The OPEN specifies a logical record length of 64. In fact, we must know beforehand what the LRL will be in reading back a file. There is no mechanism for obtaining this from the TRSDOS calls.

Reading The File Back

After a successful OPEN, the NRN (next record number) in the DCB is set to 0000H in preparation for the READ (or WRITE — the TRSDOS routines don't know which will follow). A series of READs is then done in much the same fashion as the WRITES. Each read specifies a UREC (user record area) in HL which is a video memory line.

After 16 reads, a CLOSE is done to terminate the read action. Note that in this case we knew how long the file was and simply read in 16 logical records. This is not very good programming practice with files; we really should have worked with the ERN (ending record number) and NRN (next record number) in conjunction with a discrete count of records in case there had been some programming or logic error in creating a file that had fewer records than expected. We hope you'll forgive us in this example!

The program will cause the following actions over several seconds: The screen contents (a DEBUG display is good) are written out to disk. The screen is cleared and filled with the same data from the SCREEN file. There are four distinct reads as each sector is read in, with four blocks of four lines being output rapidly to the screen after each read.

Killing a File

The KILL call is very similar to the CLOSE, except that it deletes the file name from the directory and releases the disk space used for the file to the common pool of granules. The action of the KILL can be seen by substituting a CALL 442CH in place of the CALL 4428H near WRT070 in Figure 11-14. Do a DIR command in TRSDOS after a CLOSE and after a KILL, and you will observe the KILL action.

We did not verify the physical records as we were writing them out in Figure 11-14. You have a choice of writing out each physical record without reading it back in for comparison (a "normal" WRITE) or writing out a record and reading it back in for comparison after each sector write. In my opinion, you should always verify. I'm assuming that all data being written out to disk is important to you. Although

the VERIFY takes slightly longer because a second disk operation must be performed to read in the sector for comparison, it's just good programming practice to double-check.

Change the CALL 4439H after WRT020 to CALL 443CH to perform the VERIFY write. You might want to compare the time for both a WRITE sequence and a VERIFY sequence.

— Hints and Kinks 11-6 —
How Many Disk Errors Will There Be?

(What is Truth? What is Beauty?) The floppy disk is an extremely reliable device for an electro-mechanical peripheral. If you choose not to VERIFY your writes, you'll probably not have an error in a diskette full of data. However, the overhead really is not that great when you consider the high data rates of the disk as compared to cassette operation

Using POSN

We created a sequential file in the code of Figure 11-14 by the process of writing a series of logical records in sequence. However, it's just as easy to work with random records.

In the case of a sequential file, the TRSDOS READ and WRITE routines keep pointers to the next logical record in the buffer in the DCB, along with incrementing the NRN (next record number). If we are to work with random records of a file, we must aid TRSDOS in locating the physical records by using the POSN call.

The POSN call is used to specify a logical record number in the BC register. TRSDOS then finds the proper logical record by either resetting the DCB pointers (if the logical record is in the current buffer) or reads in the sector containing the logical record and positions the pointers. This intermediate operation is necessary because there is a good chance that the random logical record is **not** in the current buffer.

Adding to a File

The POSN command is used to find the last record of an existing file so that additional records may be appended. To do this, setup the BC register pair with a record number corresponding to ERN (ending record number) + 1. The POSN will automatically find the 'end of file.' You can then do normal WRITES if you are adding to a sequential file.

Figure 11-17 shows the use of POSN in reading the first and sixteenth records back from the SCREEN file. The code here should be used **after** the SCREEN file has been created by the program shown in Figure 11-14. Two READS are done after an OPEN. Each READ is preceded by a POSN call. The first POSN specifies logical record number 7, while the second call specifies logical record number 15.

There are many subtleties involved in working with disk files, but possibly this brief introduction has been helpful in getting you started. Try experimenting with using the disk calls as described in the TRSDOS manual. The TRSDOS I/O calls may be used to create some powerful assembly-language programs that use disk a lot more efficiently than BASIC code.

```

8D0D          OD1DD          ORG          8D0DH
D011D : PROGRAM TO READ SCREEN FILE AND DEMONSTRATE POSN CALL
OD12D : CLEAR SCREEN
8DDD 21DD3C DD130 LD HL,3CDDH ;SCREEN START
8DD3 3E20 DD140 WRT04D LD A,' ' ;BLANK
8DD5 77 DD15D LD (HL),A ;STORE BLANK
8DD6 23 DD16D INC HL ;BUMP PNTR
8DD7 7C DD17D LD A,H ;GET MS BYTE
8D08 FE4D DD18D CP 40H ;TEST FOR END
8DDA 2DF7 DD19D JR NZ,WRTD4D ;GO IF NOT END
DD20D : READ BACK LOGICAL RECORDS 7 AND 15
8D0C 215D80 OD210 LD HL,BUFFER ;BUFFER LOCATION
8D0F 115D81 DD220 LD DE,WDCB ;DCB LOCATION
8D12 064D DD23D LD B,64 ;64 BYTES PER LOG RECORD
8D14 CD2444 DD24D CALL 4424H ;OPEN SCREEN FILE
8D17 28D8 DD25D JR Z,WRTD5D ;GO IF NO ERROR
8D19 F68D DD26D WRTD45 OR 8DH ;SETUP FOR ERROR MSG
8D1B CDD944 OD27D CALL 4409H ;WRITE ERROR MSG
8D1E C32D40 DD28D JP 4D2DH ;REBOOT
8D21 215D8D OD29D WRT050 LD HL,BUFFER ;BUFFER ADDRESS
8D24 115D81 OD3DD LD DE,WDCB ;DCB ADDRESS
8D27 D1D7DD OD31D LD BC,7 ;LOGICAL RECORD 7
8D2A CD4244 DD32D CALL 4442H ;POSN CALL
8D2D 20EA DD33D JR NZ,WRTD45 ;GO IF ERROR
8D2F 21D03C DD34D LD HL,3C00H ;UREC LOCATION
8D32 115D81 DD35D LD DE,WDCB ;DCB LOCATION
8D35 CD3644 DD36D CALL 4436H ;READ UREC
8D38 20DF DD37D JR NZ,WRTD45 ;GO IF ERROR
8D3A 215D8D OD38D LD HL,BUFFER ;BUFFER ADDRESS
8D3D 115D81 OD39D LD DE,WDCB ;DCB ADDRESS
8D4D D1DFDD OD4DD LD BC,15 ;LOGICAL RECORD 15
8D43 CD4244 DD41D CALL 4442H ;POSN CALL
8D46 2DD1 DD42D JR NZ,WRT045 ;GO IF ERROR
8D48 21CD3F DD43D LD HL,3FCDH ;LAST SCREEN LINE
8D4B 115D81 DD44D LD DE,WDCB ;DCB ADDRESS
8D4E CD3644 DD45D CALL 4436H ;READ RECORD
8D51 2DC6 DD46D JR NZ,WRTD45 ;GO IF ERROR
8D53 115D81 DD47D WRTD7D LD DE,WDCB ;DCB ADDRESS
8D56 CD2844 OD48D CALL 4428H ;CLOSE FILE
8D59 2DBE DD49D JR NZ,WRTD45 ;GO IF ERROR
8D5B 18FE DD5DD WRTD80 JR WRT080 ;LOOP HERE
O1DD DD510 BUFFER DEFS 256
815D 53 DD520 WDCB DEFM 'SCREEN'
      43 52 45 45 4E 20 2D 2D
      2D 2D 2D 2D 2D 2D 2D 2D
      2D 2D 2D 2D 2D 20
8174 DD DD53D DEFB DDH
8175 20 DD54D DEFM ' '
      2D 2D 2D 2D 2D 2D 2D
DD0D DD55D END
DD0D0 TOTAL ERRORS

```

Figure 11-17. Use of POSN

SECTION III

Larger Assembly-Language Projects

Chapter Twelve

Assembly-Language Design, Coding, and Debugging

In this chapter, we'll follow a typical assembly-language programming project from beginning to end. It'll be a "medium-sized" project, one that might well take a professional programmer a month or more to complete in all phases. We'll describe the trials and tribulations of that programmer and illustrate the steps required in any large assembly-language job. (The punch line of this chapter: "For you see, my friends, that professional programmer was ME!")

This is a story about Cal Coder, a programmer/analyst at GIGO Software, Inc. GIGO is one of the smaller companies producing software packages for the Radio Shack TRS-80 microcomputers.

The Inception Phase

Cal had just walked into his office at GIGO when his phone rang. "Cal, this is Paul . . . Can you drop in for a second? I think we might have something you'll be interested in."

Having served in the Sea Scouts, Cal knew a direct order couched in polite terms. He started toward his boss's office.

After the usual small talk, Paul explained, "The guys in Marketing have come up with this idea about a new program for the TRS-80 — the ads start tomorrow, and we've already sold a hundred copies. Can you get right on

it? Here're the notes on it," he said, handing Cal a matchbook cover with an ad for "Earn Big Money in Your Spare Time Programming" on one side and several scribbled notes on the other.

Looking over the notes, Cal saw that there was a need for a Morse Code Generator program that would act as a Morse code instructor or playback prerecorded Morse code messages. Marketing had done some analysis in Butte, Montana that indicated potential sales of hundreds of copies. It was up to Cal to come up with a specification to further define the project.

Research

"Do you know anything about Morse code?" Cal asked Ted, his officemate and fellow-programmer.

"Is it anything like a Hamming code?" Ted replied, puzzled.

Hints and Kinks 12-1

Hamming Code

Ted was making a pun at Cal's expense. A Hamming code is a special code frequently used in telemetry, but not often used in computer processing. Missing bits in data can be regenerated by analyzing the remaining bits. Obviously, the overall data transmission rate is reduced by inclusion of data bits for reconstruction.

Few enough bits are ''dropped'' in computers to make a code such as this unnecessary in normal applications. Parity bits or other check bits do provide some verification of data, which is usually sufficient.

"Well, hams use it, but I don't think so," Cal replied. "I guess I'll have to do some research on it before I write that spec for the Morse Code Generator I'm working on."

"Why change your approach now?" Ted asked with a leer. Cal threw a carton of C90 cassettes at him.

Later, Cal had dug up most of what he needed to know about Morse code. It was basically a series of long and short pulses representing the letters of the alphabet, digits, punctuation marks, and some special characters. The number of pulses varied. A frequently used letter like "e" consisted of one short pulse, called a "dot" or "dit." A "t" consisted of one long pulse called a "dash" or "dah." Less frequently used letters, digits, and punctuation marks consisted of longer combinations of dots and dashes. A "p", for example, was represented by "dot dash dash dot," while a "5" was "dot dot dot dot dot."

Cal also located the specifications on the standard lengths for dots and dashes and on the spacing. A dot was one unit long, while a dash was three units long. The space between dots or dashes was one unit long, the space between characters was three units, and the space between words was five units long.

Cal verified his findings with some amateur radio friends and got more data. He supplemented this research with some articles in *BYTE*, *80-Microcomputing*, and other computer magazines, and after several days had done enough research to feel he knew more than enough to write the spec.

—Hints and Kinks 12-2—
Programming Periodicals

Here's a list of some current publications which would have general programming information:

73 Magazine	R
80 Microcomputing	T
ACM Computing Surveys	P
BYTE	H
Communications of the ACM	P
Computer Design	P
Computer Magazine (IEEE)	P
Creative Computing	H
Datamation	P
Dr. Dobb's Journal	H
Eighty US Journal	T
Ham Radio	R
Interface Age	H
Kilobaud Microcomputing	H
Personal Computing	H
Popular Electronics	E
QST	R
Radio Electronics	E

E=Electronics oriented, some computer topics

H=Hobbyist computer magazine

P=Professional magazine

R=Amateur radio, some computer topics

T=TRS-80 topics exclusively

The Preliminary Specification

The spec he produced is shown in Figure 12-1. It's a preliminary operational specification that does not show anything about the actual implementation of the program. As a matter of fact, in looking over the spec, we might ask ourselves whether it is possible to produce such a program. Certainly we know the TRS-80 can produce messages on the screen, can generate audio tones out of the cassette output, and read character strings from the keyboard. But can it generate Morse code characters at 60 words per minute? Is that too fast for even assembly language?

SPECIFICATION: GIGO SOFTWARE PRODUCTS MORSE CODE GENERATOR PROGRAM, MORSE

General Description

This assembly-language software package is a TRS-80 Model I program that will run in systems with 16K of RAM or greater. It will generate international Morse code through the cassette output. Code generated will be either random code characters for practice purposes or a user-defined string of Morse code characters. Speeds of operation are user-defined and may be from 3 to 60 words per minute.

Loading Procedure

The MORSE package is loaded from disk by entering the command, MORSE, while in TRSDOS command mode. The MORSE program will be loaded and execution will start immediately.

The MORSE package is loaded from cassette by entering the BASIC command mode. After the > prompt, the following sequence is entered to load and execute the MORSE program:

```
>CMD "T"  (Turn off real-time-clock in Disk BASIC)
>SYSTEM  (Enter Monitor mode)
*? MORSE (Load cassette tape file MORSE)
*?/      (Start execution after successful load)
```

Operating Instructions

After loading, MORSE will clear the screen and position the cursor to the "home" position at the upper left corner of the screen. It will also establish a communication area on the bottom three lines of the screen as shown in Figure 1.

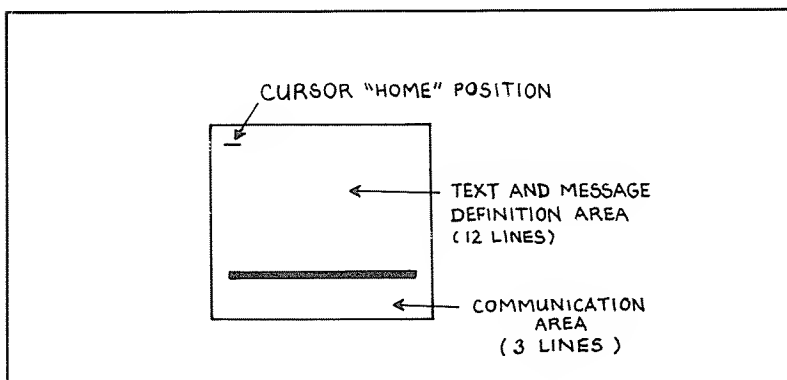
Pressing CLEAR at any time will "reset" MORSE and cause the following line to be displayed:

MORSE

CHAR=SEND CHARACTER SHIFT 0-9=SEND MSG N
SHIFT R=SEND RANDDM
SHIFT D=DEFINE MSG SHIFT S=DEFINE SPEED
SHIFT P,N=PRINT OR ND

The above information is also displayed after loading.

**Figure 12-1. Specification
for MORSE**



**Figure 1. MORSE Display
Characteristics**

Normal operation: MORSE normally operates in the "send message" mode, where a single key stroke will send a character or one of 10 messages. Pressing a 0 through 9 key with a SHIFT will send a predefined message 0 through 9, while pressing the SHIFT R keys will send a string of random characters. If the message 0 through 9 has not been defined, no action will occur. Transmission of message 0 through 9 will continue at the current speed of operation until the end of the message is reached. At the end of the message, MORSE will await the next command. If a (R)andom string is being sent, transmission will continue indefinitely until the CLEAR key is pressed.

At the same time a character, message, or random

characters are being sent, the message will be displayed on the screen and optionally printed on the system line printer (see "Line Printer Operation" below).

Define Message Mode: The define message mode is entered by entering SHIFT D. After the SHIFT D has been pressed, MORSE will respond by the line

```
DEFINE MESSAGE MODE. ENTER MESSAGE  
# 0-9:
```

The user must then enter a valid message number 0 through 9. If a valid message number is not entered, the line

```
INVALID MESSAGE NUMBER. MUST BE 0 - 9
```

is displayed, and the user must reenter the number. The user may simply press CLEAR to exit the Define Message Number mode.

Once the message number has been entered, MORSE displays the line

```
ENTER MESSAGE. TERMINATE BY ENTER
```

The user can now enter the Morse code message to be stored as message number N. Valid characters for the message are the alphabetic characters A through Z, numeric digits 0 through 9, period, comma, question mark, slash (/), and several special characters. The special characters are a dash, for "break" (-....-); space, for word space; semicolon for "error" (.....); and equals for "end message" (.-.-.). At the end of the message, the user presses the ENTER key. Invalid characters are ignored and are neither entered as part of the message, displayed, nor printed.

The size of the message is limited to 256 bytes per message. MORSE checks the amount of memory used and will generate the error message

```
MORE THAN 256 CHARACTERS. 256 ACCEPTED
```

if there is not enough space to store the current message. Only the first 256 characters of the current message will be stored if this condition occurs!

Deletion of Messages: Messages may be deleted by entering the Define Message mode and pressing the ENTER key for message text. This action releases any previously allocated memory area to the system message area.

Define Speed Mode: the Define Speed mode is entered by entering the keys SHIFT S during normal operation. MORSE responds with

SET SPEED MODE. ENTER SPEED 3 TO 60 WPM:

The user must now enter an appropriate speed for message transmission, followed by an ENTER. If a valid speed is entered, the MORSE speed is set to that value. If an invalid speed is entered, MORSE will display the message

INVALID SPEED. MUST BE 3 TO 60

The user must then reenter a correct speed.

The "default" speed of MORSE is 3 words per minute.

Line Printer Operation: Messages may be optionally printed on the system line printer by pressing the keys SHIFT P during normal operation. Pressing the keys SHIFT N disables line printer operation. The system line printer must be capable of responding at the desired speeds. A speed of 60 words per minute, for example, is about 300 characters per minute or 5 characters per second. If the printer has a long "carriage return" time without buffering data, some characters may be lost by certain types of slow printers.

Random Character Generation: During this mode, a sequence of pseudo-random (repeatable) characters are generated. Comparison of code practice copy and the sequence may be performed by examination of

line printer or display output.

Audio Output: Audio output from MORSE is through the cassette "AUX" output which would normally connect to the system cassette recorder. Audio output may be recorded directly on blank cassette tape and replayed, or the AUX output may be connected to an external audio amplifier.

Amateur Radio Output: Amateur radio operators may use the audio output to key transmitters through appropriate circuitry of their own design. GIGO SOFTWARE, INC. can assume no liability in interfacing to such an application.

In producing the specification, Cal drew upon his experience with similar types of programs. He did some preliminary computations that verified speeds of 60 words per minute could be obtained with no problems. One of the formulas he found gave the words per minute speed as —

$$\text{WPM} = \text{dots per second} * 2.4$$

Working from this, Cal deduced that the fastest **dot time** would be 50 milliseconds, or 50,000 microseconds, or about 10,000 instruction times. He concluded that this was slow for assembly language. However, this was a crucial stage. There were many other problems that may not be visible at this point, as we shall see.

Hints and Kinks 12-3

Instruction Times

Instruction times in the TRS-80 range from 4 T cycles to 23 T cycles. A T cycle (T state) is 1/4 of a clock cycle and is used in Zilog literature to define instruction speeds. The clock rate of the TRS-80 is about 1.77 megahertz, making a T cycle about .564 microseconds and instruction times about 2.255 microseconds to 51.9 microseconds (!). The nominal instruction time is about 9 T cycles or about 5 microseconds, since many instructions take 4, 7, 8, 9, and 10 T cycles.

Program Design

Cal's next task was to produce a complete **implementation specification**. This would be a specification that would discuss technical considerations of implementing the program. This step is often ignored for short programs and sometimes ignored for major programs involving many man years of work, much to the dismay of some companies.

The implementation spec includes flow charts of all program code and may also include a text description of portions of the program. It may also define and specify **system tables**.

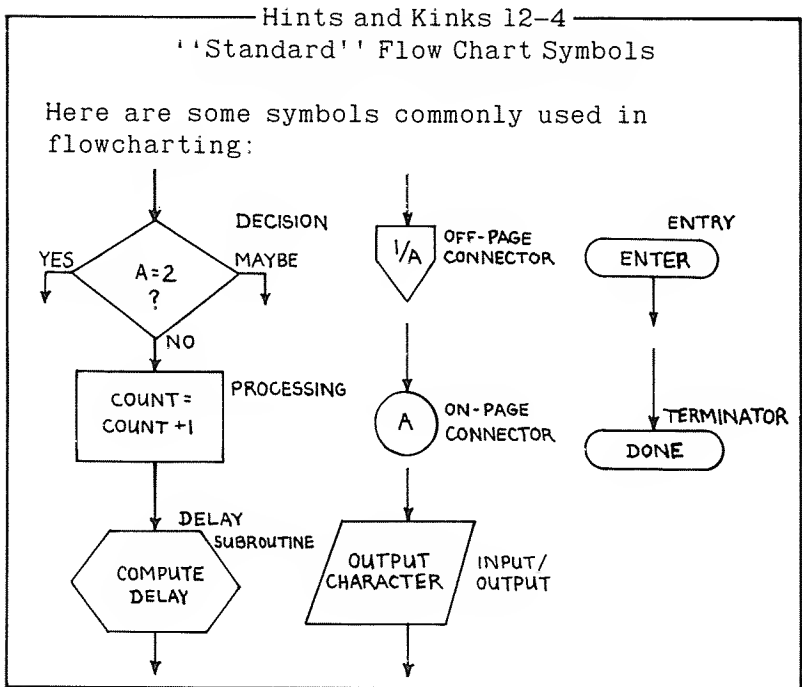
Cal was too experienced a programmer to simply start coding the program. He knew that even for small jobs a set of flow charts uncovered many **logic errors** and helped him visualize the **structure** of the program.

As for program structure, Cal usually used a combination of "top-down" and "bottom-up" design. The "top-down" approach dictated the program be designed from major functions downward, finally culminating in the lowest level modules or subroutines. The "bottom-up" approach used the opposite tack: the program was first coded at the lowest level of routines, and then progressively more complicated routines were designed.

Cal already had a preliminary spec that represented the "top-down" operation of the program. His job was now to implement the spec by defining program functional **modules** that would perform the logical functions of the program. A module is a collection of code performing a well-defined specific function with a set of inputs to produce a well-defined set of outputs. He knew that parts of the design would utilize old code, perhaps intact, such as a DELAY subroutine to delay a specified time. He also knew that other parts would be new code that he would have to write from scratch.

"Do you have your flow-chart template?" Cal asked Ted. "I've got to get this design done before I slip the schedule again."

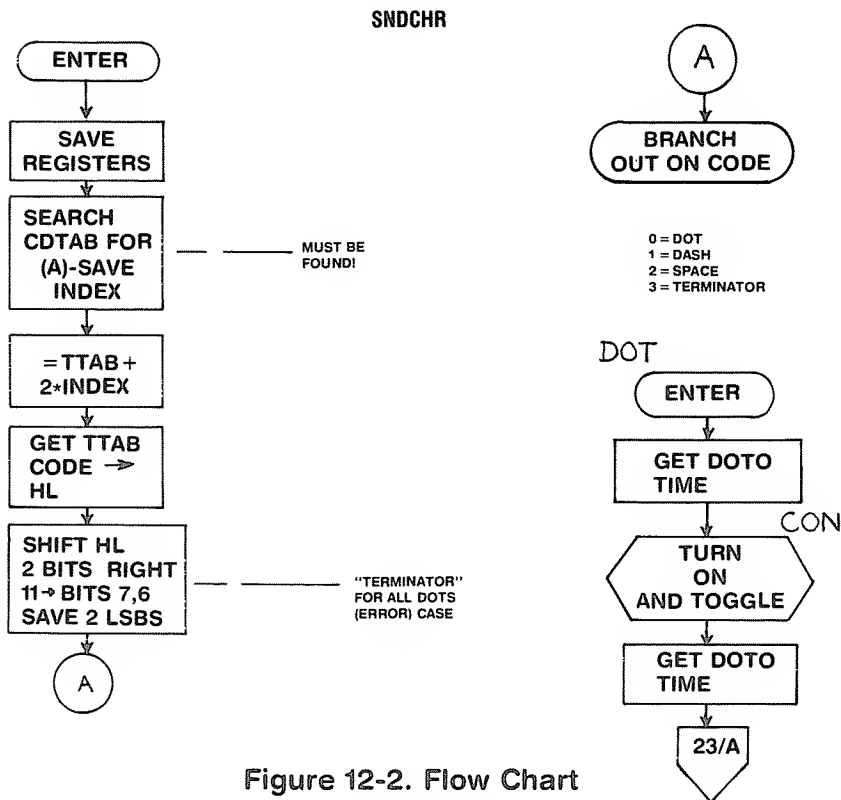
"Sorry, I used mine to scrape up the TRSDOS diskette I left out in the sun yesterday. The template melted, too."



Cal found a spare template and set to work. He did not simply produce page after page of flowcharts, but spent a great deal of time contemplating various aspects of the design, occasionally referring to reference materials or notes.

Several days later, Cal was done with the flow charts. A sample of what he produced is shown in Figure 12-2. Although there are several standard flowcharting symbols, Cal used those which his company had established as **their** standard. They were similar to those used by most other programmers in defining programming operations — a rectangular box for "processing," a

diamond for a decision point, a “subroutine” symbol, on and off page **connectors**, etc. The important thing to Cal was that he produce a shorthand version of the program he was going to code in standard notation.



“Would you take a look at these when you get a chance?” Cal asked Ted.

Cal’s company had a policy of **design reviews** for major jobs. After a programmer had finished specifying and designing a large program, he would distribute copies of the implementation spec and flow charts to the other programmers in the department. Then, after each had had a chance to look them over, a general meeting would be held to review the design.

The meetings produced their share of nit-picking comments but also brought to light omissions or errors in most designs.

Since this was a smaller program, Cal was not obligated to hold a design review, but he did want to have his officemate review the flowcharts to catch any obvious errors.

"Oh, oh!" Ted exclaimed while looking over one page of the flowcharts.

"What's the matter?"

"I don't think you've thought this through! You've got the program picking up a character from the keyboard, converting it to Morse code, outputting the code through the cassette port, and then going back to pick up the next character."

"Right — what's wrong with that?"

"Well, that means that the next character typed can't be output 'til the last one is done!"

"So?"

"If you do that, the user has to wait instead of typing normally. He'd have to type a character, wait for the tone to stop, type the next . . ."

"Oh," Cal groaned, "I see what you mean. I need a **buffered** input that'll pick up keys even during output of characters!"

Hints and Kinks 12-5

The Problem

If the input was not buffered, each character would have to be output before a new character was read from the keyboard. This doesn't seem annoying in theory, but in practice it would make for Morse code output that would have longer spaces than normal between characters and would inhibit the operator from typing freely.

"You didn't need this job, anyway!" Ted said with a grin and ducked as a copy of *TRS-80 Assembly-Language Programming* went flying by his head.

Cal did some revision and came up with a new set of flowcharts. After another review by Ted, he was ready to do the coding.

In the process of flowcharting, Cal defined several levels of modules. They are shown in Figure 12-3. The reader may want to refer to Figure 13-8 to see how they relate to the actual code of the program

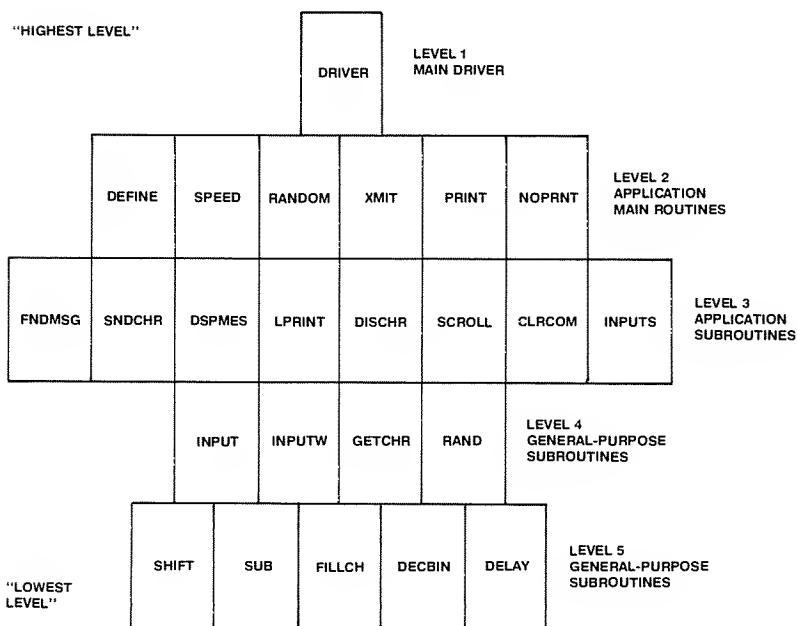


Figure 12-3. Typical Program Modules

Coding

Cal's flowcharts were such that he could have handed them to a junior programmer for coding. On larger jobs, there might be teams of programmers working on the project, each one coding his portion of the job after the implementation specification and flowcharting were done and reviewed. On this job, however, Cal did his own coding.

With the flowcharts and spec done, the coding went very rapidly. Cal didn't use regular coding sheets, although many of the programmers did. He wrote his code in pencil on quadrille pads since he would be entering the code on the system himself. The company maintained data entry operators that would take coding on coding sheets and enter it into the system for assembly, but Cal preferred to avoid the **local queues** on smaller jobs and enter the code himself.

Cal was very familiar with the Z-80 instruction formats, but remembered the problems he had had when he first started working with Z-80 code. It had taken him some time to get familiar enough with the instruction formats that he could automatically put down the correct form. Another problem had been the large number of instruction types available. Now, he rarely consulted the manual for instruction formats and operation.

One thing that Cal made certain of was to put a comment on virtually every line of the program. He knew they would be invaluable not only months later, but would help **days** later in interpreting some of the instructions.

Finally, the coding was done. Cal used his own TRS-80 in the office to enter the code and assembled the program. On the first assembly, he had several dozen assembly errors, most of them produced by a misspelled label. Correcting the errors, Cal reassembled, and after three edits and reassemblies got a "clean" assembly.

He compared the assembly listing with his original code in

great detail, and then put the original code away in what he called his "coward's nook," a place he put material he thought he'd never use again but was afraid to throw away.

Desk Checking

Cal's next task was **desk checking**. In this step of producing the MORSE program, Cal went over the assembly listing in minute detail. He compared it with the flowcharts for **logical errors** and looked at each section of code for such things as registers destroyed in subroutines when they should not have been and errors in using the stack.

On this last point, Cal paid particular attention to detail. He made certain there was a POP for every PUSH and a RET for every CALL, or at least a proper adjustment of the stack pointer. More than once he had had his program "gobbled up" by a hungry stack.

On some of the code Cal "played computer" and used a paper and pencil to record the actual operation of a subroutine, just as the Z-80 would do. He drew a line with all Z-80 registers as a heading, assumed some initial entry data for the subroutine, and then laboriously went through instruction by instruction, decrementing registers, adding results, and counting loops. One of his sessions for the SCROLL subroutine is shown in Figure 12-4.

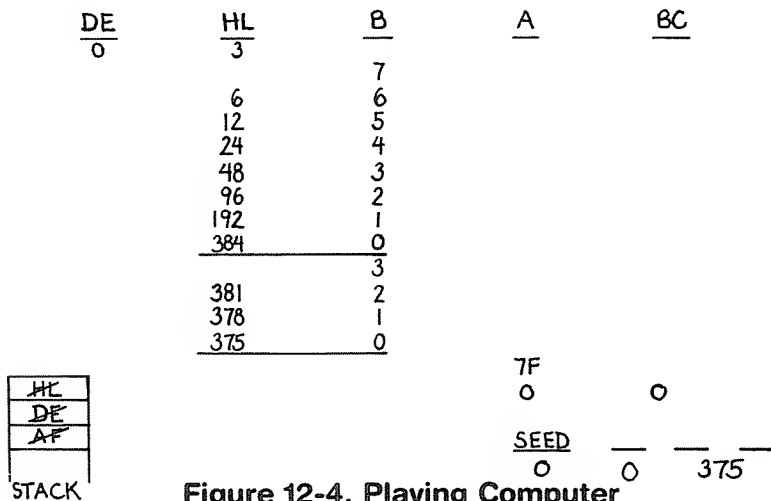


Figure 12-4. Playing Computer

As a result of the desk checking, Cal edited and reassembled twice again. The first time picked up some fairly important errors he had made in logic, while the second time cleaned up some minor problems he had found on the desk check after the first reassembly. Cal finally felt confident enough in the code to try **on-line** debugging.

— Hints and Kinks 12-6 —
How Much Desk Checking?

How much desk checking is necessary? If you were debugging on a large system with many programmers and limited access, you'd want to do a great deal of desk checking. On your own TRS-80, however, do enough to keep from being forced to reassemble more than about once every four hours of debugging time. This usually means a couple of passes through all the code initially with fairly close scrutiny. Find a comfortable trade-off between time spent in reassembling and time spent desk checking.

"Ok, Ted, I'm going to run this turkey. Want to see it work the first time?"

Ted raised his eyebrow and said, "I **almost** had a program run the first time. Of course, it was only three instructions long . . . "

Debugging

Actually, at this point, many of Cal's debugging problems were over. He had written a good implementation spec, flowcharted the program, had a one-man review, and had done comprehensive desk checking. The debugging task at this point would probably only be for minor oversights.

Cal loaded MORSE and then created a disk file version by the DUMP command. Loading it in again, he used Disk DEBUG to start execution at location 8000H. The screen filled with "@@@@@@@" and the disk "rebooted."

"That wasn't quite the result I expected," muttered Cal as Ted turned back to his desk.

Cal debugged the program by using the "binary search" on errors we discussed in earlier chapters. By the end of the day, he had patched the program to the point where it ran through the initialization fairly successfully and even output a single character on the cassette audio. The next error he found, however, was one that was not easily patched, and he was forced to reedit and reassemble.

It took Cal several more days of debugging to get to the point where he could find no more **known errors**. At that point he turned to his next task in the production cycle.

Comprehensive Testing

GIGO, Inc. was a software house producing software for the TRS-80 and other computers. The company had long ago learned to perform comprehensive tests on their software, rather than releasing it prematurely. Of course, there was constant pressure from Sales for a volume of interesting new products, but the Programming Department manager was firm in his resolve to eliminate as many bugs as possible before releasing the product (he was also the son-in-law of the chairman of the board).

Cal spent several days drawing up a **test plan** that would exercise as many of the features of MORSE as possible. It included such things as a check to see that all characters would be properly output, that all speeds could be utilized, that all **limit** conditions (such as 256 characters per message) worked, and that there was an even distribution of random characters.

When he finished, he had a formalized test plan on paper. This was submitted to his manager, Paul, and kept in the program file. Cal then went down the test plan step by step and tested each item. He found several minor discrepancies and several **human factor** improvements, such as the length of time that error messages were displayed on the screen.

After the last test item had been cleared up, Cal reassembled the "last version" of the program, went through the test plan again and found no errors. He knew that it was entirely possible that lurking in the depths of the MORSE program were small bugs that might grow their way to the surface when a user in Chillicothe decided to transmit four error codes together while using the printer during a quarter moon. He also knew that it was virtually impossible to eliminate all bugs except by continued testing over great lengths of time. However, he was confident that most users would be very happy with the program.

With some trepidation, Cal walked into Paul's office and said, "Well, here it is, Paul — the final version of MORSE!"

"Great, Cal. By the way, I'm glad you're here. I just got in a request to develop a TRS-80 program to learn Tic-Tac-Toe. . . ."

Final Clean-Up

Cal's work on MORSE wasn't quite done. He revised the final specs on the program and filed the specs, listing, source code, and working program. He knew the importance of this because when he first joined the company, he had to take two existing programs and correct errors that had been discovered. The programmer who had written them, "No Comment" Garigan, had left the company to become a tour guide in Pismo Beach.

Garigan's code contained, as his nickname implied, virtually no comments on any lines. His flowcharts were incomplete, his design specs nonexistent. Since that time, Cal had made it a special point to be as thorough in documentation as possible. He knew it was entirely possible that one of those tiny bugs might become the "Monster That Ate the TRS-80" and that he himself might be forced to correct his own code.

No Resemblance to Programmers Living or Dead

Although the scenario above is fictitious, it is an attempt to show an idealized situation in program design and development. Most programs are developed under less formalized steps, and many delete such important steps as flowcharting and final testing. In developing your own assembly-language programs, you'll adopt the methods that work for you, but it may actually take **less** development time to go through the procedures defined above . . . so some programmer a year from now won't nickname **you** "No-Comment" Garigan!

In the following two chapters, we'll show the flowcharts and listings for two large assembly-language programs: one for Morse Code Generation and one for a Tic-Tac-Toe program that "learns." These should clearly represent some of the elements of **programming style** that we've discussed.

Chapter Thirteen

A Morse Code Generator Program

In this chapter we'll look at the design and implementation of a program to generate Morse code by audio tones sent to the cassette output port. The program may be used for code practice, or it may be used for amateur radio applications to record and playback messages and normal keyboard characters at speeds of 3 to 60 words per minute. One of the features of the program is that it is fully **buffered** — messages being typed on the keyboard may "overrun" the actual Morse code output and may be dozens of characters ahead.

This program was designed as a typical assembly-language application for this book to illustrate some of the concepts discussed Chapter 12 on large program design and implementation. Since we'll also discuss the code in the program, we'll be tying together many of the coding concepts discussed earlier.

General Specification

The general specification for the program is shown in Figure 12-1. (Only the name has been changed from "MORSE" to "MORG".) The program has basically three modes. The first mode is keyboard output. Characters typed on the keyboard will be output as audio-frequency Morse code characters from the cassette output (the plug that normally attaches to the cassette recorder "AUX" jack). A small, inexpensive amplifier can be used to play the resulting output through a small speaker. As characters are output, they are displayed on the screen and can be printed on the system line printer.

The second mode of operation is integrated with the first. You can generate predefined messages of up to 256 characters in length by a single keystroke. Up to ten messages can be defined and generated, and you can intersperse messages and normal text. As characters are output, they are displayed on the screen and optionally printed on the system line printer. A typical message might be defined as message 5 and read CQ CQ CQ DE WDGCTY WDGCTY WDGCTY K. Every time you pressed the "SHIFT, 5" keys, this message would be generated.

The third mode of operation functions independently of the other two. With Random mode, you can generate an endless string of pseudo-random characters for code practice. As the characters are output to the cassette port, they are displayed on the screen and can be printed on the system line printer. The speed of operation of any of these modes can be defined to be 3 to 60 words per minute.

Operation

After loading, MORG clears the screen, draws a line near the bottom to define a "communication area," and outputs a title message of

MORG

CHAR=SEND CHARACTER SHIFT 0-9=SEND MSG N
SHIFT R=SEND RANDOM SHIFT D=DEFINE MSG
SHIFT S=DEFINE SPEED SHIFT P,N=PRINT OR NO

You can now choose one of the options in the title message. To define the speed, you press SHIFT S and enter a speed value of 3 to 60 words per minute. After the speed is defined, the title message is again printed. To set simultaneous printing on the system line printer, you press SHIFT P. The line printer will now echo the Morse characters displayed on the screen. To disable the printer at any time, you press SHIFT N. Both actions terminate by display of the title message. To define a message, you press SHIFT D. The define message mode is now entered

and will prompt you to input a message number of 0 through 9 and to enter the text of the message. Any number of characters may be entered for the text up to 256. You terminate the message by an ENTER key press, bringing you back to the title message display.

To output characters and messages, you can type in characters from the keyboard. Each time you type a legitimate character, there is an output in Morse code from the cassette output at the currently defined rate of speed. Illegitimate keys are ignored. Each time you enter a SHIFT (followed by 0 through 9) the previously defined corresponding message will be output in its entirety. If no message has been defined, nothing will be output. As keyboard text or messages are output, the text is simultaneously displayed on the screen and output to the system line printer if Print has been set.

If you have chosen the Random mode by SHIFT R, a continuous stream of pseudo-random text will be output to the cassette port, simultaneously displayed on the screen, and (optionally) printed on the system line printer. The speed will be at the currently defined rate of speed. The Random mode simulates normal text by inclusion of spaces at regular intervals.

Hints and Kinks 13-1

Simulating Text

Text is simulated here by including spaces in the CTAB, the 128-byte table of random characters. There are 18 spaces out of 128 characters, making the frequency about a space every six characters. The program also checks to make certain that two spaces are not sent consecutively.

Legitimate characters are the alphabetic characters A through Z, digits 0 through 9, comma (,), period (.), slash (/), question mark (?), and three special characters. The

three special characters are a dash for a "break" (—....—), an equals for end of transmission (.—.—.), and a semicolon for "error" (.....). All other characters are ignored.

General Design

The general design considerations or research into the methods of Morse code generation can be broken down into the following areas:

- Characteristics of Morse code
- Generation of audio tones
- Keyboard read routines
- "Buffering"
- Conversion to Morse code characters
- Message storage and search

We'll discuss each of these areas, the alternative methods that could have been used, and the final method that was adopted in MORG.

Characteristics of Morse Code

Morse code consists of a series of **dots** and **dashes** to represent alphabetic, numeric, and special characters as shown in Figure 13-1. A dot is defined as a short burst of an audio tone or other signal, while a dash is a longer tone. The combinations of dots and dashes used are generally based on the frequency of the letter in normal text. An "e" for example is one dot, while a "q" is dash, dash, dot, dash. The special character question mark (?) is dot, dot, dash, dash, dot, dot.

A	.-.	N	-. .	0	- - - - - / - - - .
B	-	O	- - - -	1	. - - - -
C	- . - .	P	. - . .	2	. . - - -
D	- . .	Q	- . - . -	3	. . . - -
E	. . .	R	. - .	4 -
F	S	. . .	5
G	- . - .	T	- . .	6	-
H	U	. . -	7	-
I	. .	V	. . . -	8	-
J	. - - - -	W	. - - -	9	-
K	- . -	X	- . . -
L	. - . .	Y	- . - -	,	-
M	- -	Z	- . . .	?

"T"	"R"	"S"	"8"	"0"
-	- - - - -	- - - - -

Figure 13-1. Morse Code Symbols

The relationship of dots, dashes, spaces between dots and dashes, and spaces between words is shown in Figure 13-2. A dot time is the basic unit, a dash time is three units, the time between dashes and dots is one unit; the time between characters is three units; and the time between words is five units. When an audio tone is used for the Morse code, the tone is on during the dot or dash time and off during the spaces as shown.

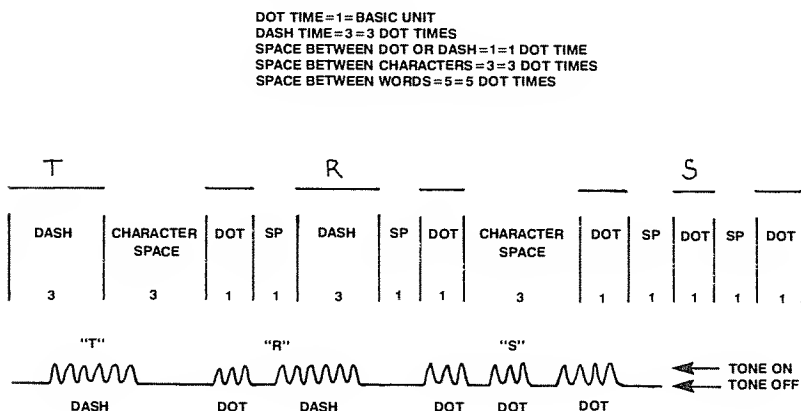


Figure 13-2. Morse Code Spacing and Audio

A slow speed for Morse code transmission is 5 words per minute. This is the speed used for the amateur radio "novice" code qualifying test. Average speeds are fifteen words per minute, while high-speed "brass pounders" send at rates of 35 words per minute or greater.

An average word is 5 characters, making the number of characters per minute $5 \times \text{WPM}$ (Words Per Minute). Fifteen WPM is therefore about 75 characters per minute, or about 1.25 characters per second. The average number of dots and dashes in a character is hard to define. One formula relates the number of WPM to dots per second:

$$\text{WPM} = \text{DPS} \times 2.5$$

This would mean that 10 dots per second are equivalent to 25 WPM. We'll use this formula for our analysis in MORG.

In MORG, the code speed may vary between 3 and 60 WPM, allowing for a very slow user and a very fast one. The number of dots per second for 3 WPM is 1.2; the number of dots per second for 60 WPM is 24. Since a string of dots consists of a dot **on** time and a dot **off** time, this makes the dot duration:

$$\text{Dot duration in milliseconds} = 1200/\text{WPM}$$

For example, the duration of a dot at 3 WPM is $1200/3$, or 400 milliseconds, while the duration of a dot at 60 WPM is $1200/60$, or 20 milliseconds. On the surface, it appears that an assembly-language program should have no problem in achieving these speeds, since 20 milliseconds, the most stringent case, represents 4000 **instruction times** at 5 microseconds per instruction ($5 \text{ microseconds} \times 4000 = 20000 \text{ microseconds}$ or 20 milliseconds).

The assembly-language program's main concern is to turn on an audio tone, leave it on for lengths of time ranging from 20 milliseconds (60 WPM dot) to 1.2 seconds (3 WPM dash), and turn it off.

Generation of Audio Tones

As we know from Chapter 10, you can easily use the TRS-80 to generate a wide range of audio tones. In this case, we need to generate only one frequency, unless we choose to make the pitch variable.

Hints and Kinks 13-2

Generating Tones

You can accomplish tone generation here by 'toggling' the cassette output latch on and off. You output alternating 01s and 10s to the cassette latch address OFFH. The 01 causes a 'high' level, and the 10 causes a 'low' level. A zero reference level would be created by 00.

The audio tone will be on for a dot or dash on time and off for inter-character or inter-word spacing. Our only problem would be the shortest duration dot-time that will have to be handled. Does this represent a duration during which we can generate a tone or is the duration so short that we can't **toggle** the cassette latch on and off to produce a tone?

The shortest duration dot-time from above is 20 milliseconds. During that 20 milliseconds, we must toggle the cassette latch on and off several times. If we did it one time, on and off, the **period** of the tone produced would be 20 milliseconds.

As the **frequency** of such a tone is 1/period, or 50 cycles per second, there seems to be no problem in the tone. Fifty cycles per second is too low for both comfortable listening and amplification on an inexpensive amplifier. If we tried for 500 cycles per second (500 **hertz**), we would be toggling the cassette latch ten times on and off for the shortest duration dot, which would be fine.

Keyboard Read Routines

We know from Chapter 7 that we can hand-tailor a keyboard read routine to read any key and convert it to whatever character we want. In most cases here, we'll be doing a "straight" conversion into the ASCII code associated with the key. However, we will have to convert some keys into special codes to represent characters for "end of message," "error," and the like. Still, we should have no problem in the conversion.

Buffering Keyboard Input & Polling the Keyboard: Problems

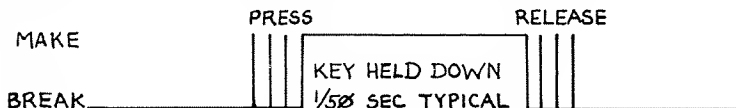
There **will** be a problem in the area of keyboard speed, however! The specification allows for **buffering** keyboard input. This means we can be typing at a faster rate than the data is being transmitted in Morse code audio. We'll have to store the characters in some sort of large buffer, depending upon how many characters ahead we wish to store.

This also means that as the character is being transmitted, we must **poll** the keyboard to see if the next character is being input. If it is, we must **debounce** the character and store it in the buffer. What other problems do we face at this point?

One major problem is this: If we use a time-delay-loop in order to generate the on/off times that toggle the cassette latch to produce an audio tone, we can't be going out during the generation of dots and dashes to **poll** the keyboard!

Debouncing

Debouncing works like this: When a key is depressed, it bounces up and down in the space of fractions of milliseconds, making and breaking the contact. When a key is released, the same make and break occurs. Keys are held down for approximately 1/50th of a second. The make/break diagram looks roughly like this:



"Rollover" allows the next key to be detected when a previous key is still being held down. MORG does not have rollover, but looks for the next key 100 milliseconds or so from the last. This would allow for typing speeds up to 120 words per minute.

Debouncing detects the first "make" and then waits 100 milliseconds to bypass bounce on the press and release.

Scanning (or polling) the keyboard normally takes 50 to 100 milliseconds (1/20th to 1/10 second) because the character has to be debounced. If we wait this long to debounce a character that's being input, it'll extend the dot or dash time for high-speed transmission (for instance, a dot-time for 20 WPM is 60 milliseconds). If we delay 50 milliseconds to debounce, the dot-time changes to 110 milliseconds! This is clearly unacceptable.

What to do, what to do On the one hand, we need to be able to read the keyboard during transmission of characters to eliminate a tedious "wait until complete before reading keyboard" condition. On the other hand, it appears we can't debounce the character because it will affect the dot and dash times.

Buffering & Polling: Some Solutions

Is there any other way to **time-out** the bounce? If there were, we could rush out for a fraction of a millisecond or so during dots, dashes, or spaces and check to see if a key were pressed. If a key **were** pressed, we could quickly decode the key and store it in a buffer, start some sort of timer, and then go back to the dot/dash/space generation. We could then periodically check for an elapsed time of 50 milliseconds or so. If 50 milliseconds had elapsed, we could reset the timer and go back and repeat the process again.

A disk system would certainly let us perform the timing since it counts in increments of 25 milliseconds. But we may not have a **real-time clock** if we don't have a disk system. Is there some way to **emulate** the real-time-clock function?

One way would be to establish a counter inside our program. This software counter would count in increments of 1 millisecond. As we'll be using a subroutine to time the cycles of the audio tone, we could use the delay count for the delay subroutine to increment the counter by the number of milliseconds delayed. For a continuous stream of characters, this would give us a fairly accurate count of elapsed time.

However, we don't have a continuous stream of characters, do we? Sure, some times we've buffered a large number of characters, and they keep on being generated at the current speed. But what about the case where the user is doing a "hunt and peck" at the keyboard? If the counter is only incremented during the time delays for the audio tones, it certainly won't be a record of elapsed time.

Well, we'll be looping and waiting for that next character from the keyboard and that loop should be at a fairly constant speed. We can establish an increment of the software millisecond counter every "n" times through the loop. This should work for the rough elapsed time needed for debounce. After all, it doesn't matter too much whether we delay 50 milliseconds or 100 milliseconds for a

maximum typing speed of 60 WPM (or five characters per second). Is this the best way to handle this problem? Probably not, but it's **one** way given the constraints of the system — no real time clock, no **interrupts**, and a need for buffering.

To minimize the time required to rush out and poll the keyboard, we'll need a fast keyboard scan routine, one that determines quickly if a character is there, and if not, returns to the calling routine. It'll also have to test for the debounce delay time from the software counter.

Buffering

What about the problem of **buffering**? How do we store the characters as they are read in and then transmit them?

To do this we'll need a buffer of a certain number of bytes. As the characters are read from the keyboard, they are stored in the next **slot** of the buffer. The cassette tone output will have to somehow test the contents of the buffer to see if there is any new character available for output.

If the operator is a very slow typist and the code speed is slow enough, this buffer will be filled with one character and then immediately emptied. However, if the operator is fast and the code speed is slow, the buffer may fill up rapidly with characters that are being input too fast to be handled.

What about the size of the buffer? If the program runs for any length of time, we'd need a very large buffer. A better way to do this would be to establish a **circular buffer** based on the worst-case input and output speed. We'll assume that the operator can't get more than 256 characters ahead before he gets confused! This should be adequate for most of us.

The circular buffer is used as shown in Figure 13-3. A new character goes into the next slot. A pointer to the next slot is then incremented by one. If we reach the end of the buffer, the pointer is set back to the beginning of the

buffer. A second pointer points to the “next character available.” If the two pointers are equal, all characters have been used. “Overrun” is possible if the number of unused characters exceeds 256.

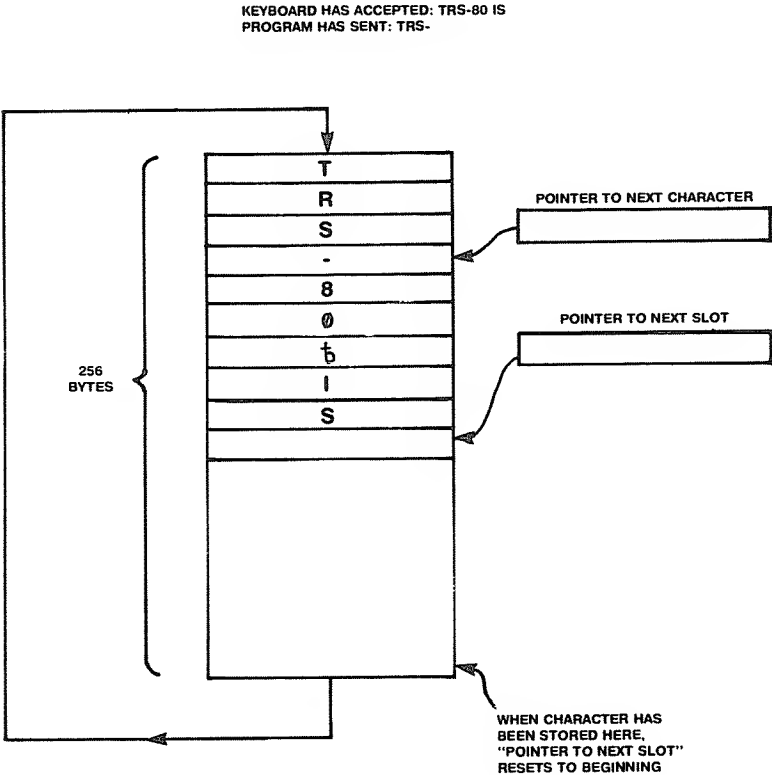


Figure 13-3. Circular Buffer

Conversion to Morse Code Characters

How will a given ASCII character be converted to a Morse code character? We have 26 alphabetic characters, ten digits, blank, and seven special characters to be concerned with. The Morse code combinations are completely unrelated to ASCII codes or to any other “neat” method of finding the proper combination of dots and dashes.

The implication is that instead of a formula or **algorithmic** means to produce the Morse code, we need to translate an ASCII character into a Morse code character by a **table lookup**. One Morse code configuration will be in the table for each legitimate ASCII character.

What about storage of the dots and dashes in the table? Each character we're concerned with is represented by six or fewer dots or dashes, except for the "error" code. It looks like we could use six codes for each character. One way to do this would be to store a fixed eight-byte entry for each character, with 0 representing a dot, 1 a dash, 2 a space, and -1 a terminator (or we could pack two codes in each byte).

After some thought, we came up with this: Each combination of dots and dashes is held in 16 bits, or two bytes. There are eight **fields** in the two bytes. Each field contains a two-bit code. A code of 00 represents a dot, a code of 01 represents a dash, a code of 10 is a dot space, and a code of 11 is a terminator. The 8 fields read from **right to left**. Each combination ends with a terminator field of 11, allowing up to seven dots, dashes, or dot spaces, plus a terminator.

This scheme is shown in Figure 13-4.

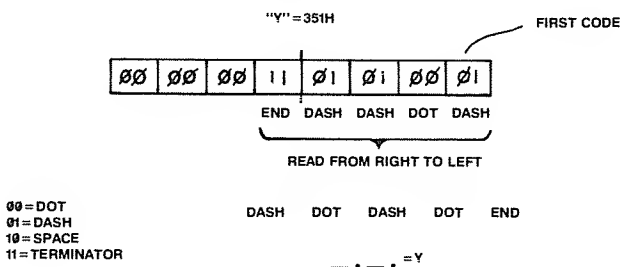


Figure 13-4. Dot, Dash, Space Coding

The special case of eight dots ("error code") is represented by two bytes of zeroes and is detected in the program.

Message Storage and Search

Messages are stored as ASCII character strings. A large buffer capable of holding $10 \times 257 + 1$ or 2,571 characters (and ten one-byte "headers") is established. A minus one is used as a terminator for the last message in the buffer. Unused spaces in the buffer are filled with minus one bytes.

All bytes in the buffer are either valid ASCII characters, minus ones, or the binary digits 0 through 9. A binary digit 0 through 9 marks the **header** of a message string. The length of the string is determined by the next occurrence of a digit, or a minus one.

Messages are put into the buffer as they are defined. A search is made for the header number to find a specified message. If a message is redefined, the old message is first deleted by "moving up" the remaining buffer into the space previously occupied by the message. This approach is shown in Figure 13-5.

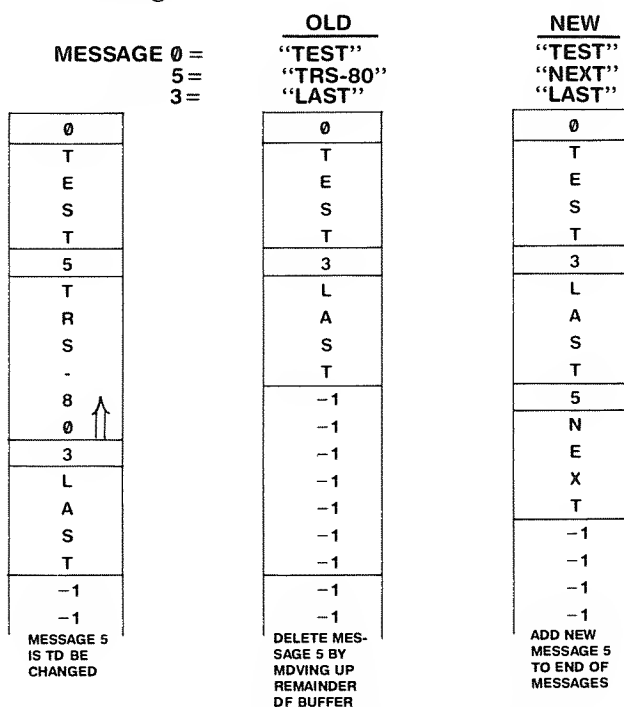


Figure 13-5. Message Storage

Implementation

Modules

MORG is implemented as a series of five levels of modules, shown in Figure 13-6. The top level of modules is the main **driver** of the program. The bottom level of modules are the most rudimentary subroutines of the program.

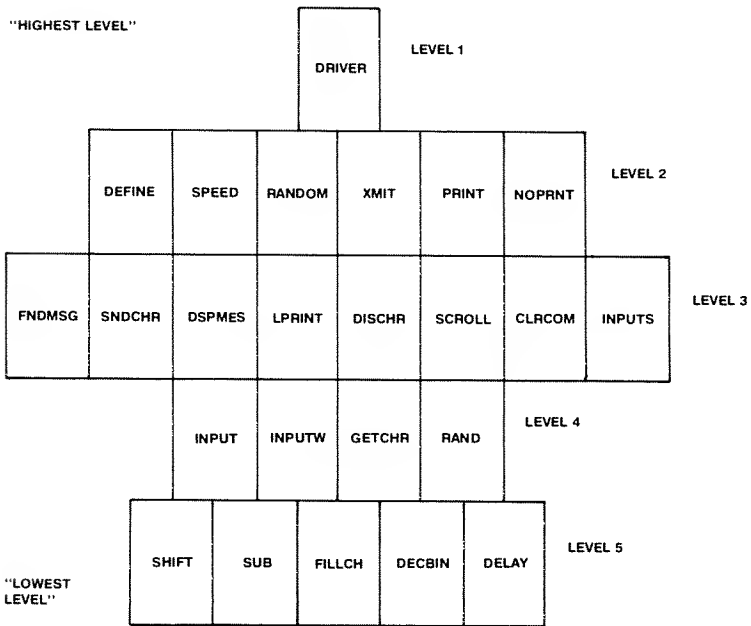


Figure 13-6. MORG Modules

The modules are generally dedicated to a particular well-defined function related to the MORG application, such as "finding a message number" or "clearing the communications area." Each module is generally in the form of a subroutine with one entry point and one exit point. The lowest level of subroutines are general-purpose subroutines, useful for many applications.

Figure 13-7 shows the modules and their interconnections. Each module generally calls another module by a CALL, with a set of parameters in the CPU registers. Higher level modules usually call lower-level modules, although some modules call other modules on the same level.

Hints and Kinks 13-4
Notes on Figure 13-7

The module interconnection diagram of Figure 13-7 reveals some interesting facts about the structure of MORG. The level one driver not only calls every level two routine but also does applications-related processing by calling level three routines, which are special purpose routines for the Morse code application. Level two routines also make heavy use of level three routines.

If we see only one connection (dot) along any horizontal connection to a module, it might indicate that the modules should be incorporated as 'in-line' code. This occurs for SHIFT, SUB, and RAND.

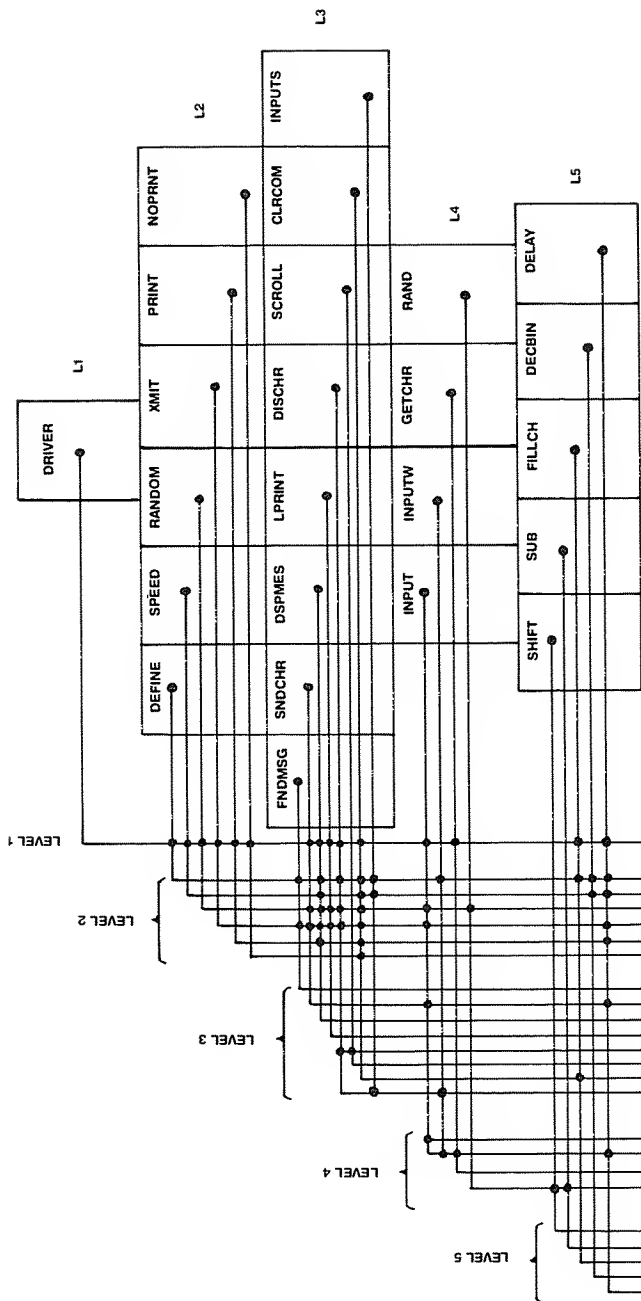


Figure 13-7. MORG Module Interconnections

To find the modules called by any particular module, follow the horizontal line from the module to the extreme left. When the line turns downward, read the lower level module called by referencing the connection dots. The arrangement of the modules is duplicated in the program listing, that is, higher-level modules appear first, followed by lower-level modules.

Tables, Buffers, and Variables

Refer to the MORG listing, Figure 13-8. Here is a set of variables, buffers, and tables the program uses to define system status, hold text, and facilitate conversion. The variables are held in the working storage area of the program, near the end. For the most part, their use is explained by the comments associated with each one.

```

8000      00100      ORG      8000H
          00110      :MORG-0820
          00120      :*****MORSE CODE GENERATOR PROGRAM*****
          00130      :
          00140      :
          00150      :*****SYSTEM EQUATES*****
          00160      :
3C00      00170      SCREEN EQU      3C00H      ;START OF VIDEO DISPLAY
3C40      00180      LINE1 EQU      SCREEN+64      ;SECOND LINE
3EC0      00190      LINE11 EQU      SCREEN+704      ;TWELFTH LINE
3F00      00200      LINE12 EQU      SCREEN+768      ;THIRTEENTH LINE
3F40      00210      LINE13 EQU      SCREEN+832      ;FOUTEENTH LINE
3F80      00220      LINE14 EQU      SCREEN+896      ;FIFTEENTH LINE
3FC0      00230      LINE15 EQU      SCREEN+960      ;SIXTEENTH LINE
0002      00240      ENTER EQU      2      ;ENTER CHARACTER
0001      00250      CLEAR EQU      1      ;CLEAR CHARACTER
0064      00260      DBDEL EQU      100      ;DEBOUNCE DELAY IN MS
0065      00270      DBDELP EQU      DBDEL+1      ;DEBOUNCE DELAY+1 MS
000A      00280      MDEL EQU      10      ;MAIN LOOP DELAY IN 1/10 MS
03C0      00290      SPEEDF EQU      960      ;FINAGLE FACTOR FOR SPEED
          00300      :
          00310      :*****MORSE EXECUTIVE*****
          00320      :
8000      00330      START DI          ;DISABLE INTERRUPTS
8001      315994      LD      SP, TOPS      ;SET STACK POINTER
8004      11EA88      LD      DE, MBUF      ;MESSAGE BUFFER ADDRESS
8007      010B0A      LD      BC, 2571      ;2571 BYTES
800A      3EFF        LD      A, 0FFH      ;-1 FOR FILL
800C      CD8385      CALL      FILLCH      ;FILL MESSAGE BUFFER
800F      21D204      LD      HL, 1234      ;INITIALIZE RANDOM # SEED
8012      22E185      LD      (SEED), HL
8015      212E16      LD      HL, 5678
8018      22E385      LD      (SEED+2), HL
801B      3E20        LD      A, ' '      ;BLANK CHARACTER
801D      11003C      LD      DE, SCREEN      ;START OF SCREEN
8020      010004      LD      BC, 1024      ;# OF BYTES
8023      CD8385      CALL      FILLCH      ;CLEAR SCREEN
8026      3E8F        LD      A, 08FH      ;ALL ON GRAPHICS CHAR
8028      11003F      LD      DE, LINE12      ;LINE 12
802B      014000      LD      BC, 64      ;# OF BYTES
802E      CD8385      CALL      FILLCH      ;DRAW LINE
8031      21003C      LD      HL, SCREEN      ;SCREEN START
8034      22E985      LD      (CURCUR), HL      ;INITIALIZE CURRENT CURSOR
          00530      : REENTER HERE FOR MOST FUNCTIONS
8037      216400      LD      HL, DBDEL      ;DEBOUNCE DELAY IN MS
803A      CDC085      CALL      DELAY      ;DELAY
803D      216500      LD      HL, DBDELP      ;MINIMUM DELAY +1

```

8040	22E085	00570	LO	(TSLC),HL	:INITIALIZE FOR NEXT CHARACTER
8043	21F592	00580	LO	HL,IBUF	:INPUT BUFFER ADDRESS
8046	22F085	00590	LD	(IBUFL),HL	:INITIALIZE INPUT BUFFER PTRS
8049	22F285	00600	LO	(IBUFL),HL	
804C	2AE985	00610	LO	HL,(CURCUR)	:GET CURRENT CURSOR
804F	22E885	00620	LD	(LSTCUR),HL	:SAVE
8052	CDFB83	00630	CALL	CLRCOM	:CLEAR COMMUNICATIONS AREA
8055	21F485	00640	LO	HL,MSG1	:INITIAL MESSAGE
8058	01403F	00650	LO	BC,LINE13	:LINE 13
805B	CD6083	00660	CALL	OSPMES	:OUTPUT MESSAGE
805E	2AE885	00670	LO	HL,(LSTCUR)	:GET OLO CURSOR
8061	22E985	00680	LO	(CURCUR),HL	:RESTORE
		00690	:	REENTER HERE	:DURING TRANSMISSION OF CHARACTERS OR MSGS
8064	1600	00700	MOR020	LO	D,0
					:INITIALIZE MINOR COUNT
8066	C02984	00710	MOR021	CALL	INPUT
					:GET RESPONSE
8069	C01585	00720	CALL	GETCHR	:GET CHARACTER
806C	200F	00730	JR	NZ,MOR022	:GO IF PRESENT
806E	14	00740	INC	0	:BUMP MINOR COUNT
806F	7A	00750	LO	A,0	:GET COUNT
8070	FE0A	00760	CP	MLOEL	:TEST FOR 1 MS
8072	20F2	00770	JR	NZ,MOR021	:GO IF NOT 1 MS
8074	2AE085	00780	LO	HL,(TSLC)	:GET TIME
8077	23	00790	INC	HL	:BUMP BY 1 MS
8078	22E085	00800	LO	(TSLC),HL	:RESTORE
807B	18E7	00810	JR	MOR020	:CONTINUE
807D	B0	00820	MOR022	OR	B
					:MERGE SHIFT BIT
807E	21CC80	00830	LO	HL,FTAB+FTABS-1	:FUNCTION TABLE ENO ADO
8081	060F	00840	LO	B,FTABS	:FUNCTION TABLE SIZE
8083	BE	00850	MOR025	CP	(HL)
					:TEST FOR FUNCTION
8084	2825	00860	JR	Z,MOR030	:GO IF POUND
8086	2B	00870	DEC	HL	:POINT TO NEXT FUNCTION
8087	10FA	00880	OUNZ	MOR025	:CONTINUE
8089	E67F	00890	AND	7FH	:RESET BIT 7
		00900	:	NOT FUNCTION	:HERE - TRANSMIT A SINGLE CHARACTER
808B	320C85	00910	LO	(TMP1),A	:STORE IN TEMP BUFFER
808E	3EFF	00920	LO	A,OFFH	:1
8090	320085	00930	LD	(TMP1+1),A	:STORE TERMINATOR
8093	210C85	00940	LO	HL,TMP1	:ADDRESS OF "MESSAGE"
8096	7E	00950	MOR027	LO	A,(HL)
					:GET NEXT CHARACTER
8097	FE20	00960	CP	'	:TEST FOR NON-ASCII
8099	FA6480	00970	JP	M,MOR020	:GO IF ENO OF MESSAGE
809C	C0BE83	00980	CALL	OISCHR	:DISPLAY CHARACTER
809F	CD8183	00990	CALL	LPRINT	:PRINT IF REQUIRED
80A2	C0E082	01000	CALL	SNOCHR	:SEND CHARACTER
80A5	23	01010	INC	HL	:POINT TO NEXT
80A6	CD2984	01020	CALL	INPUT	:TEST FOR INPUT
80A9	18EB	01030	JR	MOR027	:CONTINUE SENDING
80AB	48	01040	MOR030	LO	C,B
					:INIOEX+1 NOW IN C
80AC	0D	01050	DEC	C	:ADJUST FOR INIOEX
80A0	0600	01060	LO	B,0	:INDEX NOW IN BC
80AF	CB21	01070	SLA	C	:2*INIOEX NOW IN BC
80B1	0021C080	01080	LO	IX,BTAB	:BRANCH TABLE
80B5	0D09	01090	A00	IX,BC	:POINT TO BRANCH
80B7	006601	01100	LO	H,(IX+1)	:GET MSB OF ADDRESS
80BA	D06E00	01110	LO	L,(IX)	:GET LSB OF ADDRESS
80B0	E9	01120	JP	(HL)	:BRANCH OUT
		01130	:		
		01140	:	FUNCTION TABLE	
		01150	:		
80BE	C4	01160	FTAB	DEFB	'D'+80H
					:OEFINE MESSAGE
80BF	03	01170	OEFB	'S'+80H	:OEFINE SPEED
80C0	02	01180	OEFB	'R'+80H	:TRANSMIT RANDOM
80C1	B0	01190	OEFB	'0'+80H	:TRANSMIT MESSAGE 0
80C2	B1	01200	OEFB	'1'+80H	:1
80C3	B2	01210	OEFB	'2'+80H	:2
80C4	B3	01220	OEFB	'3'+80H	:3
80C5	B4	01230	OEFB	'4'+80H	:4
80C6	B5	01240	OEFB	'5'+80H	:5
80C7	B6	01250	OEFB	'6'+80H	:6
80C8	B7	01260	OEFB	'7'+80H	:7
80C9	B8	01270	OEFB	'8'+80H	:8
80CA	B9	01280	OEFB	'9'+80H	:9
80CB	D0	01290	DEFB	'P'+80H	:SET PRINT
80CC	CE	01300	DEFB	'N'+80H	:RESET PRINT
000F		01310	FTABS	EQU	\$(FTAB
					:SIZE OF FUNCTION TABLE
		01320	:		
		01330	:	BRANCH TABLE	
		01340	:		

```

80CD EB80      01350 BTAB      DEFW      DEFINE      ;DEFINE MESSAGE
80CF 9181      01360          DEFW      SPEED        ;DEFINE SPEED
80D1 F081      01370          DEFW      RANDOM        ;TRANSMIT RANDOM
80D3 3782      01380          DEFW      XMIT         ;TRANSMIT MESSAGE 0
80D5 3782      01390          DEFW      XMIT         ;
80D7 3782      01400          DEFW      XMIT         ;
80D9 3782      01410          DEFW      XMIT         ;
80DB 3782      01420          DEFW      XMIT         ;
80DD 3782      01430          DEFW      XMIT         ;
80DF 3782      01440          DEFW      XMIT         ;
80E1 3782      01450          DEFW      XMIT         ;
80E3 3782      01460          DEFW      XMIT         ;
80E5 3782      01470          DEFW      XMIT         ;
80E7 8B82      01480          DEFW      PRINT        ;SET PRINT
80E9 B182      01490          DEFW      NOPRNT       ;RESET PRINT
              01500          ;
              01510          ;*****DEFINE MESSAGE N ROUTINE*****
              01520          ;
80EB 2AE985    01530          DEFINE LD      HL,(CURCUR) ;GET CURRENT CURSOR
80EE 22EB85    01540          LD      (LSTCUR),HL      ;SAVE
80F1 CDFB83    01550          DEF005 CALL    CLRCOM      ;CLEAR COMMUNICATIONS AREA
80F4 21F586    01560          LD      HL,MSG5         ;DEFINE MESSAGE
80F7 01403F    01570          LD      BC,LINE13        ;LINE 13
80FA CD6D83    01580          CALL    DSPMES          ;DISPLAY DEFINE MESSAGE
80FD 0601      01590          LD      B,1              ;1 CHARACTER
80FF CD0D84    01600          CALL    INPUTS          ;GET CHARACTER
8102 C27F81    01610          JP      NZ,DEF050        ;GO IF GT 1
8105 0601      01620          LD      B,1              ;1 CHARACTER
8107 CD9085    01630          CALL    DECBIN          ;CONVERT TO BINARY
810A C27F81    01640          JP      NZ,DEF050        ;GO IF ERROR
810D 7D        01650          LD      A,L            ;MSG # NOW IN A
810E 21C987    01660          LD      HL,MSG11        ;INPUT MESSAGE
8111 01803F    01670          LD      BC,LINE14        ;LINE 14
8114 CD6D83    01680          CALL    DSPMES          ;DISPLAY MESSAGE
8117 2AEB85    01690          LD      HL,(LSTCUR)      ;GET OLD CURSOR
811A 22E985    01700          LD      (CURCUR),HL     ;RESTORE
811D F5        01710          PUSH     AF            ;SAVE MESSAGE #
811E CDC782    01720          CALL    FNDMSG         ;GET ADDRESS OF MESSAGE
8121 202A      01730          JR      NZ,DEF035        ;GO IF NO CURRENT MSG FOR #
              01740          ; CURRENT MSG FOR # IN MBUF - MUST DELETE
8123 E5        01750          PUSH     HL            ;SAVE START
8124 D1        01760          POP      DE            ;PUT IN DE
8125 23        01770          INC      HL            ;BYPASS # TO TEXT
8126 7E        01780          DEF025 LD      A,(HL)          ;GET CHARACTER
8127 FE20      01790          CP      ' '          ;TEST FOR NON-ASCII OR -1
8129 FA2F81    01800          JP      M,DEF030        ;GO IF NEXT MESSAGE
812C 23        01810          INC      HL            ;BUMP POINTER
812D 18F7      01820          JR      DEF025          ;CONTINUE
812F E5        01830          DEF030 PUSH    HL            ;END
8130 C1        01840          POP      BC
8131 E5        01850          PUSH    HL            ;SAVE NEXT AREA
8132 21F592    01860          LD      HL,ENDM        ;END OF MEMORY
8135 B7        01870          OR      A            ;CLEAR CARRY
8136 ED42      01880          SBC     HL,BC          ;# OF BYTES TO MOVE
8138 E5        01890          PUSH    HL            ;TRANSFER TO BC
8139 C1        01900          POP      BC          ;BYTE COUNT
813A E1        01910          POP      HL            ;RESTORE SOURCE
813B EDB0      01920          LDIR          ;MOVE MESSAGE DOWN
813D 21F592    01930          LD      HL,ENDM        ;END OF MEMORY
8140 B7        01940          OR      A
8141 ED52      01950          SBC     HL,DE          ;FIND # OF BYTES REMAINING
8143 E5        01960          PUSH    HL            ;TRANSFER TO BC
8144 C1        01970          POP      BC
8145 3EFF      01980          LD      A,OFFH        ;-1
8147 CD8385    01990          CALL    FILLCH        ;FILL REMAINING WITH -1S
814A CDC782    02000          CALL    FNDMSG         ;FIND FIRST -1
              02010          ; HL POINTS TO MESSAGE AREA, TOP OF STACK HOLDS MESSAGE #
814D F1        02020          DEF035 POP      AF            ;GET MESSAGE #
814E 77        02030          LD      (HL),A          ;STORE IN MESSAGE AREA
814F 23        02040          INC      HL            ;POINT TO FIRST TEXT CHAR POS
8150 0600      02050          LD      B,0            ;INITIALIZE COUNT OF CHARS
8152 CDF384    02060          DEF040 CALL    INPUTW        ;GET NEXT CHARACTER
8155 FE02      02070          CP      ENTER          ;TEST FOR ENTER CHAR
8157 CA3780    02080          JP      Z,MOR015      ;GO IF ENTER
815A 77        02090          LD      (HL),A          ;STORE IN MESSAGE AREA
815B 23        02100          INC      HL            ;BUMP POINTER
815C CDBE83    02110          CALL    DISCHR        ;DISPLAY
815F 10F1      02120          DJNZ    DEF040        ;GO IF NOT 256 CHARS
8161 2AB985    02130          LD      HL,(CURCUR)    ;GET CURRENT CURSOR

```

```

8164 22EB85 02140 LO (LSTCUR),HL ;SAVE
8167 21EB87 02150 HL,MSG12 ;"256" CHARACTERS MESSAGE
816A 01C03F 0216G LD BC,LINE15
816D CD6D83 02170 CALL OSPMES ;DISPLAY WARNING MESSAGE
8170 21D007 02180 LO HL,2000 ;2000 MILLISECONOS
8173 C0C085 02190 CALL OELAY ;DELAY 2 SECS
8176 2AEB85 02200 LD HL,(LSTCUR) ;GET 0LO CURSOR
8179 22E985 02210 LD (CURCUR),HL ;RESTORE
817C C33780 02220 JP MOR015 ;GET NEXT COMMAND
817F 21AA87 02230 DEF050 LO HL,MSG10 ;ERROR MESSAGE
8182 01803F 02240 LO BC,LINE14 ;LINE 14
8185 CD6D83 02250 CALL OSPMES ;OISPLAY ERROR MESSAGE
8188 21D007 02260 LD HL,2000 ;2000 MILLISECONOS
818B CDC085 02270 CALL DELAY ;DELAY 2 SECS
818E C3F180 02280 JP DEF005 ;TRY AGAIN
02290 ;
02300 ;*****SET SPEED ROUTINE*****
02310 ;
8191 2AE985 02320 SPEED0 LO HL,(CURCUR) ;GET CURRENT CURSOR
8194 22EB85 02330 LD (LSTCUR),HL ;SAVE
8197 CDFB83 02340 SPEED005 CALL CLRCOM ;CLEAR COMMUNICATION AREA
819A 21CB86 02350 LD HL,MSG4 ;SPEED MESSAGE
8190 01403F 02360 LO BC,LINE13 ;LINE 13
81A0 C06083 02370 CALL OSPMES ;OISPLAY SPEED MESSAGE
81A3 0602 02380 LD B,2 ;2 CHARS
81A5 C00084 02390 CALL INPUTS ;GET CHARACTER STRING
81A8 2035 02400 JR NZ,SPEED020 ;GO IF GT 2 CHARACTERS
81AA C09085 02410 CALL OECBIN ;CONVERT TO BINARY
81A0 2030 02420 JR NZ,SPEED020 ;GO IF ERROR
81AF 7D 02430 LO A,L ;GET SPEED 0-99
81B0 FE03 02440 CP 3 ;TEST FOR 3 WPM
81B2 FADF81 02450 JP M,SPEED020 ;GO IF LT 3 WPM
81B5 FE3D 02460 CP 61 ;TEST FOR 60 WPM
81B7 F2DF81 02470 JP P,SPEED020 ;GO IF GT 60 WPM
81BA 21C003 02480 LD HL,SPEED0F ;1200/WPM=DOTO TIME
81BD 4F 02490 LD C,A ;WPM LS BYTE
81BE 0600 02500 LD B,0 ;NOW IN BC
81C0 11FFFF 02510 LO DE,-1 ;QUOTIENT
81C3 B7 02520 SPEED015 OR A ;ZERO C
81C4 ED42 02530 SBC HL,BC ;OVIDE BY SUCCESSIVE SUB
81C6 13 02540 INC OE ;BUMP QUOTIENT
81C7 30FA 02550 JR NC,SPEED015 ;GO IF NOT NEGATIVE
81C9 ED53E85 02560 LO (DOTO),OE ;STORE DOT ON TIME
81CD 210000 02570 LD HL,0
8100 19 02580 A00 HL,DE ;FIND 3*DOTO
8101 19 02590 A00 HL,DE
81D2 19 02600 ADD HL,OE
81D3 22E785 02610 LO (DASHO),HL ;STORE DASH ON TIME
81D6 2AEB85 02620 LO HL,(LSTCUR) ;GET 0LO CURSOR
81D9 22E985 02630 LO (CURCUR),HL ;RESTORE
810C C33780 02640 JP MOR015 ;BACK TO DRIVER
810F 212087 02650 SPEED020 LD HL,MSG6 ;ERROR MESSAGE
81E2 01803F 02660 LO BC,LINE14 ;LINE 14
81E5 C06083 02670 CALL OSPMES ;OISPLAY ERROR MESSAGE
81E8 210007 02680 LO HL,2000 ;2000 MILLISCS
81EB C0C085 02690 CALL DELAY ;OELAY 2 SECS
81EE 18A7 02700 JR SPEED005 ;TRY AGAIN
02710 ;
02720 ;*****TRANSMIT RANDOM CHARACTERS ROUTINE*****
02730 ;
81F0 2AE985 02740 RANDOM LO HL,(CURCUR) ;GET CURRENT CURSOR
81F3 22EB85 02750 LO (LSTCUR),HL ;SAVE
81F6 CDFB83 02760 CALL CLRCOM ;CLEAR COMMUNICATION AREA
81F9 213F87 02770 LO HL,MSG7 ;RANDOM MESSAGE
81FC 01403F 02780 LO BC,LINE13 ;LINE 13 POSITION
81FF C06D83 02790 CALL DSPMES ;OISPLAY RANOM MESSAGE
8202 ED5F 02800 LD A,R ;GET REFRESH COUNT (RANOOM)
8204 32E485 02810 LD (SEED+3),A ;INITIAL SEEO
8207 3E20 02820 LO A,' ' ;BLANK
8209 32EF85 02830 LD (LASTR),A ;INITIALIZE LAST RANOOM CHAR
820C 2AEB85 02840 LO HL,(LSTCUR) ;GET OLD CURSOR
820F 22E985 02850 LD (CURCUR),HL ;RESTORE
8212 CD4285 02860 RAN010 CALL RANO ;GET RANOOM # 0-127
8215 211288 02870 LO HL,CTAB ;CHARACTER TABLE
8218 09 02880 A00 HL,BC ;POINT TO CHARACTER
8219 7E 02890 LO A,(HL) ;GET ASCII CHARACTER
821A FE20 02900 CP ' ' ;TEST FOR BLANK
821C 2008 02910 JR NZ,RAN020 ;GO IF NOT BLANK
821E E5 02920 PUSH HL ;SAVE HL

```



```

821F 21EF85 02930 LD HL, LASTR ; ADDRESS OF LAST CHAR
8222 BE 02940 CP (HL) ; TEST AGAINST LAST
8223 E1 02950 POP HL ; RESTORE HL
8224 28EC 02960 JR Z, RANO10 ; DON'T SEND 2 BLANKS
8226 CDBE83 02970 RANO20 CALL DISCHR ; DISPLAY CHARACTER
8229 CD8183 02980 CALL LPRINT ; PRINT IF REQ'D
822C CDE082 02990 CALL SNDCHR ; SEND CHAR
822F 32EF85 03000 LD (LASTR), A ; SAVE FOR NEXT COMPARE
8232 CD2984 03010 CALL INPUT ; TEST FOR CLEAR
8235 18DB 03020 JR RANO10 ; BACK TO NEXT CHAR
03030 ;
03040 ; ***** TRANSMIT MESSAGE N ROUTINE *****
03050 ;

8237 2AE985 03060 XNIT LD HL, (CURCUR) ; GET CURRENT CURSOR
823A 22EB85 03070 LD (LSTCUR), HL ; SAVE
823D CDFB83 03080 CALL CLRCOM ; CLEAR COMMUNICATION AREA
8240 216A87 03090 LD HL, MSG8 ; TRANSMIT MESSAGE
8243 C5 03100 PUSH BC ; SAVE MSG#+3*2
8244 01403F 03110 LD BC, LINE13 ; LINE 13 POSITION
8247 CD6D83 03120 CALL DSPMES ; DISPLAY TRANSMIT MESSAGE
03130 ; MESSAGE #*2 STILL IN BC

824A C1 03140 POP BC ; RETRIEVE MSG#+3*2
824B CB39 03150 SRL C ; MESSAGE #+3 IN BC
824D 79 03160 LD A, C ; MESSAGE #+3 IN A
824E D603 03170 SUB 3 ; MESSAGE # IN A
8250 CD782 03180 CALL FNDMSG ; GET ADDRESS OF MESSAGE
8253 201E 03190 JR NZ, XMT020 ; IF NONE
8255 23 03200 INC HL ; BYPASS #
8256 ED5BEB85 03210 LD DE, (LSTCUR) ; GET OLD CURSOR
825A ED53E985 03220 LD (CURCUR), DE ; RESTORE
825E 7E 03230 XMT010 LD A, (HL) ; GET NEXT CHARACTER
825F FE20 03240 CP ' ' ; TEST FOR NON-ASCII
8261 FA3780 03250 JP M, MOR015 ; GO IF END OF MESSAGE
8264 CDBE83 03260 CALL DISCHR ; DISPLAY CHARACTER
8267 CD8183 03270 CALL LPRINT ; PRINT IF REQ'D
826A CDE082 03280 CALL SNDCHR ; SEND CHAR
826D 23 03290 INC HL ; POINT TO NEXT
826E CD2984 03300 CALL INPUT ; TEST FOR CLEAR
8271 18EB 03310 JR XMT010 ; CONTINUE SENDING
8273 219587 03320 XMT020 LD HL, MSG9 ; ERROR MESSAGE
8276 01803F 03330 LD BC, LINE14 ; LINE 14 POSITION
8279 CD6D83 03340 CALL DSPMES ; DISPLAY ERROR MESSAGE
827C 21E803 03350 LD HL, 1000 ; 1000 MILLISECONDS
827F CDC085 03360 CALL DELAY ; DELAY 1 SEC
8282 2AE885 03370 LD HL, (LSTCUR) ; GET OLD CURSOR
8285 22E985 03380 LD (CURCUR), HL ; RESTORE
8288 C33780 03390 JP MOR015 ; BACK TO EXECUTIVE
03400 ;
03410 ; ***** SET PRINT ROUTINE *****
03420 ;

828B 2AE985 03430 PRINT LD HL, (CURCUR) ; GET CURRENT CURSOR
828E 22EB85 03440 LD (LSTCUR), HL ; SAVE
8291 3E01 03450 LD A, 1 ; PRINT FLAG ON
8293 32DE85 03460 LD (PRINTF), A ; STORE
8296 CDFB83 03470 CALL CLRCOM ; CLEAR COMMUNICATION AREA
8299 21B586 03480 LD HL, MSG2 ; PRINT MESSAGE
829C 01403F 03490 PRI010 LD BC, LINE13 ; LINE 13
829F CD6D83 03500 CALL DSPMES ; DISPLAY PRINT MESSAGE
82A2 21E803 03510 LD HL, 1000 ; 1000 MILLISECONDS
82A5 CDC085 03520 CALL DELAY ; DELAY 1 SEC
82A8 2AE885 03530 LD HL, (LSTCUR) ; GET OLD CURSOR
82AB 22E985 03540 LD (CURCUR), HL ; RESTORE
82AE C33780 03550 JP MOR015 ; BACK TO DRIVER
03560 ;
03570 ; NO PRINT ROUTINE
03580 ;

82B1 AF 03590 NOPRINT XOR A ; PRINT FLAG OFF
82B2 32DE85 03600 LD (PRINTF), A ; STORE
82B5 32DF85 03610 LD (LFFTF), A ; RESET FIRST TIME FLAG
82B8 2AE985 03620 LD HL, (CURCUR) ; GET CURRENT CURSOR
82BB 22EB85 03630 LD (LSTCUR), HL ; SAVE
82BE CDFB83 03640 CALL CLRCOM ; CLEAR COMMUNICATIONS AREA
82C1 21B886 03650 LD HL, MSG3 ; NO PRINT MESSAGE
82C4 C39C82 03660 JP PRI010 ; GO TO DISPLAY, DELAY
03670 ;
03680 ; ***** FNDMSG SUBROUTINE *****
03690 ; FINDS MESSAGE BY SCANNING MBUF FOR 0-9. 0-9 IS
03700 ; MESSAGE #. ASCII CHARACTERS ARE CHARACTERS TO
03710 ; BE SENT. -1 IS PADDING AT END OF MESSAGE.

```

```

03720 ;* ENTRY: (A)=MESSAGE #
03730 ;* EXIT: (HL)=POINTER TO MESSAGE IF Z OR
03740 ;* POINTER TO NEXT AVAILABLE IF NZ
03750 ;* RESET
03760 ;* ALL REGISTERS SAVED EXCEPT HL
03770 ;
82C7 C5 03780 FNDMSG PUSH BC ;SAVE BC
82C8 CDD782 03790 CALL FNDNR ;SEARCH
82CB 2807 03800 JR Z,FND020 ;GO IF FOUND
82CD F5 03810 PUSH AF ;SAVE NOT FND FLAG
82CE 3EFF 03820 LD A,OFFH ;FOR FIRST AVAILABLE
82D0 CDD782 03830 CALL FNDNR ;SEARCH
82D3 F1 03840 POP AF ;GET FLAG
82D4 C1 03850 FND020 POP BC ;RESTORE BC
82D5 2B 03860 DEC HL ;ADJUST HL
82D6 C9 03870 RET ;RETURN
82D7 21EA88 03880 FNDNR LD HL,MBUF ;START OF MSG BUFFER
82DA 010B0A 03890 LD BC,2571 ;SIZE OF MBUF
82DD EDB1 03900 CPIR ;COMPARE
82DF C9 03910 RET ;RETURN
03920 ;
03930 ;*****SEND CHARACTER SUBROUTINE*****
03940 ; SENDS A SINGLE CHARACTER AT CURRENT SPEED BY OUT-
03950 ; PUTTING A 500 HERTZ TONE TO CASSETTE.
03960 ; ENTRY: (A)=ASCII CHARACTER
03970 ; ALL REGISTERS SAVED.
03980 ;
82E0 F5 03990 SNDCHR PUSH AF ;SAVE REGISTERS
82E1 C5 04000 PUSH BC
82E2 E5 04010 PUSH HL
82E3 DDE5 04020 PUSH IX
82E5 213D88 04030 LD HL,CDTAB+CDTABS-1 ;ADDRESS OF CODE TAB END
82E8 012C00 04040 LD BC,CDTABS ;SIZE OF CODE TABLE
82EB EDB9 04050 CPDR ;SEARCH CODE TABLE
04060 ; MUST BE FOUND!
82ED C5 04070 PUSH BC
82EE DDE1 04080 POP IX
82F0 DD29 04090 ADD IX,IX ;2*INDEX IN HL
82F2 019288 04100 LD BC,TTAB ;TIMING TABLE ADDRESS
82F5 DD09 04110 ADD IX,BC ;POINT TO TIMING CHAR
82F7 DD6E00 04120 LD L,(IX) ;GET TIMING CHAR
82FA DD6601 04130 LD H,(IX+1)
82FD E5 04140 PUSH HL ;SAVE HL
82FE E1 04150 SND010 POP HL ;GET CURRENT HL
82FF 7D 04160 LD A,L ;GET NEXT 2 BITS
8300 CB3C 04170 SRL H ;ALIGN FOR NEXT NIBBLE
8302 CB1D 04180 RR L
8304 CB3C 04190 SRL H
8306 CB1D 04200 RR L
8308 CBFC 04210 SET 7,H ;SUBTLETY HERE-TERMINATOR!
830A CBF4 04220 SET 6,H
830C E5 04230 PUSH HL ;SAVE CURRENT HL
830D E603 04240 AND 3
830F 2816 04250 JR Z,DOT ;GO IF DOT
8311 FE01 04260 CP 1 ;DASH=1
8313 2820 04270 JR Z,DASH ;GO IF DASH
8315 FE02 04280 CP 2 ;DOT SPACE=2
8317 2814 04290 JR Z,SPACE ;GO IF SPACE
8319 2AE585 04300 LD HL,(DOTO) ;INTERCHAR SPACE=3
831C 29 04310 ADD HL,HL ;2*DOTO+PREVIOUS DOTO
831D CD5683 04320 CALL COFF ;DELAY 2 DOT TIMES
8320 E1 04330 POP HL ;RESET STACK
8321 DDE1 04340 POP IX ;RESTORE REGISTERS
8323 E1 04350 POP HL
8324 C1 04360 POP BC
8325 F1 04370 POP AF
8326 C9 04380 RET ;RETURN
04390 ;
04400 ; DOT HERE
04410 ;
8327 2AE585 04420 DOT LD HL,(DOTO) ;DOT ON TIME
832A CD3D83 04430 CALL CON ;TOGGLE CASSETTE OUT
832D 2AE585 04440 SPACE LD HL,(DOTO) ;DOT OFF TIME=DOT ON TIME
8330 CD5683 04450 CALL COFF ;DELAY
8333 18C9 04460 JR SND010 ;RETURN FOR NEXT NIBBLE
04470 ;
04480 ; DASH HERE
04490 ;
8335 2AE785 04500 DASH LD HL,(DASHO) ;DASH ON TIME

```

```

8338 CD3D83 04510 CALL CON ;TOGGLE CASSETTE OUT
833B 18F0 04520 JR SPACE ;DASH OFF TIME=DOT OFF TIME
      04530 ;
      04540 ; ON HERE - GENERATE 500 HERTZ TONE
      04550 ;
833D 3E01 04560 CON LD A,01 ;ON
833F 2B 04570 DEC HL ;ADJUST FOR "JR C"
8340 01FFFF 04580 LD BC,-1 ;DECREMENT
8343 EE03 04590 ONO10 XOR 3 ;TOGGLE
8345 D3FF 04600 OUT (OFFH),A ;OUTPUT TO CASSETTE LATCH
8347 E5 04610 PUSH HL ;SAVE COUNT
8348 210100 04620 LD HL,1 ;FOR 1 MS
834B CDC085 04630 CALL DELAY ;DELAY 1 MS
834E CD2984 04640 CALL INPUT ;GET POSSIBLE CHARACTER
8351 E1 04650 POP HL ;GET COUNT
8352 09 04660 ADD HL,BC ;DECREMENT COUNT
8353 38EE 04670 JR C,ONO10 ;GO IF NOT -1
8355 C9 04680 RET ;RETURN
      04690 ;
      04700 ; OFF HERE
      04710 ;
8356 EE00 04720 COFF XOR 0
8358 2B 04730 DEC HL ;ADJUST FOR "JR C"
8359 01FFFF 04740 LD BC,-1 ;DECREMENT
835C D3FF 04750 OUT (OFFH),A ;OUTPUT TO CASSETTE LATCH
835E E5 04760 OFF010 PUSH HL ;SAVE COUNT
835F 210100 04770 LD HL,1 ;FOR 1 MS
8362 CDC085 04780 CALL DELAY ;DELAY 1 MS
8365 CD2984 04790 CALL INPUT ;GET POSSIBLE CHARACTER
8368 E1 04800 POP HL ;GET COUNT
8369 09 04810 ADD HL,BC ;DECREMENT COUNT
836A 38F2 04820 JR C,OFF010 ;GO IF NOT -1
836C C9 04830 RET ;RETURN
      04840 ;
      04850 ;*****DISPLAY MESSAGE AT LOCATION N*****
      04860 ;* DISPLAYS MESSAGE AT GIVEN SCREEN POSITION. TER- *
      04870 ;* MINUTES ON NULL (ZERO). *
      04880 ;* ENTRY: (HL)=MESSAGE LOCATION *
      04890 ;* (BC)=SCREEN POSITION *
      04900 ;* ALL REGISTERS SAVED. *
      04910 ;
836D F5 04920 DSPMES PUSH AF ;SAVE REGISTERS
836E C5 04930 PUSH BC
836F E5 04940 PUSH HL
8370 7E 04950 DSP005 LD A,(HL) ;GET MESSAGE CHAR
8371 B7 04960 OR A ;TEST FOR 0
8372 2809 04970 JR Z,DSP010 ;RETURN IF DONE
8374 02 04980 LD (BC),A ;STORE CHARACTER
8375 03 04990 INC BC ;BUMP SCREEN POINTER
8376 23 05000 INC HL ;BUMP MESSAGE POINTER
8377 ED43E985 05010 LD (CURCUR),BC ;SAVE POINTER
837B 18F3 05020 JR DSP005 ;CONTINUE
837D E1 05030 DSP010 POP HL ;RESTORE REGISTERS
837E C1 05040 POP BC
837F F1 05050 POP AF
8380 C9 05060 RET ;RETURN
      05070 ;
      05080 ;*****LPRINT SUBROUTINE*****
      05090 ;* OUTPUTS CHARACTER TO SYSTEM LINE PRINTER IF PRINT *
      05100 ;* FLAG IS SET. *
      05110 ;* ENTRY: (A)=ASCII CHARACTER *
      05120 ;* ALL REGISTERS SAVED. *
      05130 ;
8381 C5 05140 LPRINT PUSH BC ;SAVE REGISTERS
8382 E5 05150 PUSH HL
8383 47 05160 LD B,A ;SAVE CHARACTER
8384 3ADE85 05170 LD A,(PRINTF) ;GET PRINT FLAG
8387 B7 05180 OR A ;TEST
8388 78 05190 LD A,B ;RESTORE A FOR POSSIBLE RTN
8389 2821 05200 JR Z,LPRO090 ;RETURN IF NOT SET
838B 3ADF85 05210 LD A,(LPFTF) ;GET LINE PRINTER 1ST TIME
838E B7 05220 OR A ;TEST
838F 2007 05230 JR NZ,LPRO10 ;GO IF NOT FIRST TIME
8391 32E085 05240 LD (CHARCT),A ;INITIALIZE CHAR COUNT
8394 3C 05250 INC A ;1 TO A
8395 32DF85 05260 LD (LPFTF),A ;SET FIRST TIME FLAG

```

```

8398 3AE085 05270 LPRO10 LO A,(CHARCT) ;GET CHARACTER COUNT
839B E61F 05280 ANO 1FH ;GET 0-31 COUNT
8390 2005 05290 JR NZ,LPRO20 ;GO IF NOT 32ND
839F 3E00 05300 LO A,00H ;CARRIAGE RETURN
83A1 CDAF83 05310 CALL LPSTAT ;TEST STATUS AND OUTPUT
83A4 78 05320 LPRO20 LO A,B ;RESTORE CHARACTER
83A5 COAF83 05330 CALL LPSTAT ;TEST BUSY AND OUTPUT
83A8 21E085 05340 LD HL,CHARCT ;CHARACTER COUNT
83AB 34 05350 INC (HL) ;BUMP CHARACTER COUNT
83AC E1 05360 LPRO90 POP HL ;RESTORE REGISTERS
83A0 C1 05370 POP BC
83AE C9 05380 RET ;RETURN
05390 ;
05400 ; LINE PRINTER STATUS AND PRINT CHARACTER SUBROUTINE
05410 ;
83AF F5 05420 LPSTAT PUSH AF ;SAVE CHAR
83B0 3AE837 05430 LPS010 LO A,(37E8H) ;GET STATUS
83B3 E6F0 05440 ANO 0F0H ;MASK OUT GARBAGE BITS
83B5 FE30 05450 CP 30H ;TEST FOR BUSY
83B7 20F7 05460 JR NZ,LPS010 ;GO IF BUSY
83B9 F1 05470 POP AF ;RESTORE CHAR
83BA 32E837 05480 LD (37E8H),A ;OUTPUT
83B0 C9 05490 RET ;RETURN
05500 ;
05510 ;*****DISPLAY CHARACTER SUBROUTINE*****
05520 ; OUTPUTS ONE CHARACTER TO CURRENT CURSOR POSITION
05530 ; ON SCREEN. MOVES CURSOR TO NEXT POSITION UNLESS
05540 ; LAST CHARACTER POSITION OF LINE 11. IF LATTER,
05550 ; SCROLLS UP FIRST.
05560 ; ENTRY: (CURCUR)=CURRENT CURSOR POSITION
05570 ; (A)=CHARACTER TO BE OUTPUT
05580 ; ALL REGISTERS SAVED.
05590 ;
83BE C5 05600 OISCHR PUSH BC ;SAVE REGISTERS
83BF E5 05610 PUSH HL
83C0 2AE985 05620 LO HL,(CURCUR) ;GET CHARACTER POSITION
83C3 77 05630 LO (HL),A ;STORE CHARACTER
83C4 01FF3E 05640 LO BC,3C00H+767 ;LAST CP OF LINE 11
83C7 23 05650 INC HL ;BUMP CURSOR
83C8 22E985 05660 LD (CURCUR),HL ;STORE
83CB B7 05670 OR A ;RESET CARRY
83CC E042 05680 SBC HL,BC ;TEST FOR LAST
83CE 2003 05690 JR NZ,OIS010 ;RETURN IF NO SCROLL
83D0 CD0683 05700 CALL SCROLL ;SCROLL UP
83D3 E1 05710 DIS010 POP HL ;RESTORE REGISTERS
83D4 C1 05720 POP BC
83D5 C9 05730 RET ;RETURN
05740 ;
05750 ;*****SCROLL SCREEN SUBROUTINE*****
05760 ; SCROLLS LINES 1-11 UP TO LINES 0-10. FILLS LINE 11
05770 ; WITH BLANKS.
05780 ; ENTRY: NO PARAMETERS
05790 ; ALL REGISTERS SAVED.
05800 ;
83D6 F5 05810 SCROLL PUSH AF ;SAVE REGISTERS
83D7 C5 05820 PUSH BC
83D8 05 05830 PUSH OE
83D9 E5 05840 PUSH HL
83DA 11003C 05850 LO OE,SCREEN ;START OF SCREEN
83DB 21A03C 05860 LO HL,LINE1 ;LINE 1
83DC 010003 05870 LD BC,1024-256 ;# TO MOVE
83DD E0B0 05880 LDIR ;MOVE EM
83DE 11C03E 05890 LD OE,LINE11 ;START OF LINE 11
83DF 3E20 05900 LO A,' ' ;SPACE
83E0 01A000 05910 LO BC,64 ;# TO FILL
83E1 C08385 05920 CALL FILLCH ;FILL LINE
83E2 21C03E 05930 LD HL,LINE11 ;START OF LINE 11
83E3 22E985 05940 LO (CURCUR),HL ;RESET
83E4 E1 05950 POP HL ;RESTORE REGISTERS
83E5 01 05960 POP OE
83E6 F8 C1 05970 POP BC
83E7 F9 F1 05980 POP AF
83FA C9 05990 RET ;RETURN
06000 ;
06010 ;*****CLEAR COMMUNICATION AREA SUBROUTINE*****
06020 ; CLEARS SYSTEM COMMUNICATION AREA
06030 ; ENTRY: NO PARAMETERS
06040 ; ALL REGISTERS SAVED.
06050 ;

```

```

83FB F5      06060 CLRCOM  PUSH  AF          ;SAVE REGISTERS
83FC C5      06070      PUSH  BC
83FD D5      06080      PUSH  DE
83FE 3E20    06090      LD     A, ' '        ;BLANK
8400 11403F  06100      LD     DE,LINE13    ;START OF TEXT AREA
8403 01C000  06110      LD     BC,192      ;3 LINES WORTH
8406 CD8385  06120      CALL  FILLCH       ;FILL WITH BLANKS
8409 D1      06130      POP   DE          ;RESTORE REGISTERS
840A C1      06140      POP   BC
840B F1      06150      POP   AF
840C C9      06160      RET              ;RETURN
06170 ;
06180 ; *****INPUT STRING SUBROUTINE*****
06190 ; INPUTS STRING OF CHARACTERS AT CURRENT COMMUNICA-
06200 ; TION AREA. TERMINATED BY ENTER.
06210 ; ENTRY: (B)=MAXIMUM NUMBER
06220 ; (CURCUR)=CURRENT CURSOR POSITION
06230 ; EXIT: (B)=ACTUAL NUMBER INPUT
06240 ; (HL)=FIRST CHARACTER LOCATION
06250 ; NZ IF GT MAXIMUM NUMBER
06260 ; Z IF LE MAXIMUM NUMBER
06270 ; ALL REGISTERS SAVED EXCEPT HL,BC,A
06280 ;
840D 2AE985  06290 INPUTS  LD     HL,(CURCUR) ;CURRENT CURSOR POSITION
8410 E5      06300      PUSH  HL          ;SAVE
8411 04      06310      INC   B           ;BUMP MAXIMUM
8412 0E00    06320      LD     C,0         ;INITIALIZE COUNT OF CHARS
8414 C5      06330 INSO10  PUSH  BC         ;SAVE COUNTS
8415 CDF384  06340      CALL  INPUTW      ;GET CHARACTER
8418 C1      06350      POP   BC         ;RESTORE COUNTS
8419 FE02    06360      CP     ENTER      ;TEST FOR DONE
841B 2809    06370      JR     Z,INSO30    ;GO IF ENTER
841D 0C      06380      INC   C           ;BUMP CHARACTER COUNT
841E CDBE83  06390      CALL  DISCHR      ;DISPLAY
8421 10F1    06400      DJNZ  INSO10      ;GO IF NOT MAXIMUM
8423 3EFF    06410      LD     A,OFFH     ; -1 TO A
8425 B7      06420      OR     A          ;RESET Z FLAG
8426 E1      06430 INSO30  POP   HL         ;RETRIEVE START
8427 41      06440      LD     B,C        ;GET CHARACTER COUNT
8428 C9      06450      RET              ;RETURN
06460 ;
06470 ; *****KEYBOARD INPUT SUBROUTINE*****
06480 ; IF DEBOUNCE DELAY LESS THAN ELAPSED TIME, SCANS
06490 ; KEYBOARD AND STORES POSSIBLE INPUT CHARACTER IN
06500 ; CIRCULAR INPUT BUFFER.
06510 ; ENTRY: NO PARAMETERS
06520 ; EXIT: NO PARAMETERS
06530 ; ALL REGISTERS SAVED
06540 ; CLEAR CHARACTER CAUSES RESTART AT MORO15H, SP RESET
06550 ;
8429 F5      06560 INPUT  PUSH  AF          ;SAVE REGISTERS
842A 3A7F38  06570      LD     A,(387FH)    ;ALL IN ONE SWELL FOOP
842D B7      06580      OR     A          ;TEST FOR ANY KEY
842E CAB984  06590      JP     Z,INP065    ;GO IF NONE
8431 C5      06600      PUSH  BC
8432 E5      06610      PUSH  HL
8433 2AED85  06620      LD     HL,(TSLC)    ;GET TIME SINCE LAST CHARACTER
8436 016400  06630      LD     BC,DBDEL    ;MINIMUM DELAY
8439 B7      06640      OR     A          ;RESET CARRY
843A ED42    06650      SBC   HL,BC       ;COMPARE
843C 3879    06660      JR     C,INP060    ;GO IF LT DEBOUNCE DELAY
843E 210138  06670      LD     HL,3801H    ;ROW 0 ADDRESS
8441 7E      06680 INP010  LD     A,(HL)   ;GET ROW VALUE
8442 B7      06690      OR     A          ;TEST FOR NON-ZERO
8443 2007    06700      JR     NZ,INP020   ;GO IF INPUT
8445 CB25    06710      SLA   L           ;SHIFT ROW ADDRESS
8447 F24184  06720      JP     P,INP010    ;MORE TO GO
844A 186B    06730      JR     INP060     ;RETURN
06740 ; CONVERT ROW, COLUMN TO INDEX
844C 4F      06750 INP020  LD     C,A      ;ROW VALUE
844D AF      06760      XOR   A          ;ZERO A
844E CB3D    06770 INP025  SRL   L         ;SHIFT ADDRESS
8450 3804    06780      JR     C,INP035    ;GO IF DONE
8452 C608    06790      ADD   A,8         ;ROW*8
8454 18F8    06800      JR     INP025     ;CONTINUE
8456 06FF    06810 INP035  LD     B,OFFH   ;COLUMN COUNT
8458 04      06820 INP040  INC   B         ;BUMP COUNT
8459 CB39    06830      SRL   C         ;SHIFT ROW VALUE
845B 30FB    06840      JR     NC,INP040   ;CONTINUE UNTIL 1 BIT

```

```

845D 80      06850      ADD      A,B      ;ROW*8+COL
845E 4F      06860      LD        C,A      ;TRANSFER TO C
           0687D      ; FIND TABLE ENTRY
845F 0600    0688D      LD        B,0      ;NOW IN BC
8461 21BB84  06890      LD        HL,KBTAB ;ADDRESS OF LOOK UP TABLE
8464 D9      069DD      ADD      HL,BC    ;POINT TO VALUE
8465 7E      06910      LD        A,(HL)   ;GET VALUE
8466 B7      06920      OR        A      ;TEST CHARACTER
8467 284E    0693D      JR        Z,INP06D ;GO IF NOT VALID
8469 F5      06940      PUSH     AF      ;SAVE CHARACTER
846A 3A8038  06950      LD        A,(3880H) ;GET SHIFT
846D 0F      0696D      RRCA      ;ALIGN TO BIT 7
846E 47      06970      LD        B,A    ;PUT IN B
846F F1      0698D      POP       AF      ;RESTORE CHARACTER
8470 FE2F    0699D      CP        '/ '    ;TEST FOR SLASH
8472 2804    070DD      JR        Z,INP052 ;GO IF SLASH
8474 FE2D    07D1D      CP        '- '    ;TEST FOR MINUS
8476 20D6    07D2D      JR        NZ,INP055 ;NEITHER
8478 CB78    0703D      BIT        7,B    ;TEST FOR SHIFT
847A 28D2    0704D      JR        Z,INPD55 ;GO IF LOWER CASE
847C CBE7    0705D      SET      4,A    ;/ TO ? OR - TO =
847E FED1    0706D      INP055  CP      CLEAR ;TEST FOR CLEAR
848D 2D18    0707D      JR        NZ,INPD57 ;GO IF NOT CLEAR
8482 315994  07D8D      LD        SP,TOPS ;RESET STACK, DROP REGS
8485 2AE985  0709D      LD        HL,(CURCUR) ;GET LAST CURSOR POSN
8488 D14D3F  071DD      LD        BC,LINE13 ;START OF COMM AREA
848B B7      0711D      OR        A      ;CLEAR CARRY
848C ED42    0712D      SBC      HL,BC    ;COMPARE
848E FA9784  0713D      JP        M,INP056 ;GO IF IN TEXT AREA
8491 2AEB85  0714D      LD        HL,(LSTCUR) ;GET TEXT AREA PNTR
8494 22E985  0715D      LD        (CURCUR),HL ;RESET CURSOR POSN
8497 C33780  0716D      JP        INP056  JPC MOR015 ;RETURN TO EXEC
           0717D      ; STORE CHARACTER IN CIRCULAR INPUT BUFFER
849A 2AF285  0718D      INPD57  LD      HL,(IBUFN) ;GET POINTER TO NEXT
849D B0      0719D      OR        B      ;MERGE IN SHIFT BIT
849E 77      0720D      LD        (HL),A    ;STORE THIS CHARACTER
849F 23      0721D      INC      HL      ;BUMP TO NEXT SLOT
84AD 22F285  0722D      LD        (IBUFN),HL ;STORE
84A3 01F593  0723D      LD        BC,IBUFE ;END OF INPUT BUFFER
84A6 B7      0724D      OR        A      ;RESET CARRY
84A7 ED42    0725D      SBC      HL,BC    ;TEST FOR END
84A9 2006    0726D      JR        NZ,INP059 ;GO IF NOT END
84AB 21F592  0727D      LD        HL,IBUF  ;START OF I BUFFER
84AE 22F285  0728D      LD        (IBUFN),HL ;BACK TO BEGINNING
84B1 21000D  0729D      INPD59  LD      HL,D    ;ZERO HL
84B4 22ED85  073DD      LD        (TSLC),HL ;RESET TIMER
84B7 E1      0731D      INPD6D  POP      HL      ;RESTORE REGISTERS
84B8 C1      0732D      POP      BC
84B9 F1      0733D      INPD65  POP      AF
84BA C9      0734D      RET
           0735D      ;
           0736D      ; KEYBOARD LOOK UP TABLE HERE FOR LOWER CASE
           0737D      ;
84BB 0D      0738D      KBTAB  DEFB      0      ;ROW 0 - e
84BC 41      0739D      DEFB      'ABCDEFGH'
           42 43 44 45 46 47
84C3 48      0740D      DEFB      'IJKLMNOP' ;ROW 1
           49 4A 4B 4C 4D 4E 4F
84CB 5D      0741D      DEFB      'PQRSTUVWXYZ' ;ROW 2
           51 52 53 54 55 56 57
84D3 58      0742D      DEFB      'XYZ' ;ROW 3
           59 5A
84D6 DD      0743D      DEFB      0
84D7 0D      0744D      DEFB      0
84D8 DD      0745D      DEFB      0
84D9 DD      0746D      DEFB      0
84DA 0D      0747D      DEFB      0
84DB 30      0748D      DEFB      'D1234567' ;ROW 4
           31 32 33 34 35 36 37
84E3 38      0749D      DEFB      '89' ;ROW 5
           39
84E5 00      075DD      DEFB      D      ;COLON
84E6 3B      0751D      DEFB      ':' ;"ERROR"
84E7 2C      0752D      DEFB      ',-./'
           2D 2E 2F
84EB 02      0753D      DEFB      2      ;ROW 6 - ENTER
84EC D1      0754D      DEFB      1      ;CLEAR
84ED D0      0755D      DEFB      D      ;BREAK
84EE DD      0756D      DEFB      D      ;UP ARROW

```

```

84EF DD      D757D      DEFB      D      ;OOWN ARROW
84FD DD      D758D      DEFB      D      ;LFT ARRDW
84F1 OD      D759D      OEFB      D      ;RT ARRDW
84F2 2D      D76DD      OEFM      ' '

D761D ;
D762D ;*****INPUT CHARACTER AND WAIT SUBRDUTINE*****
D763D ; INPUTS SINGLE CHARACTER FRDM KEYBDARD BY WAITING
D764D ; UNTIL CHARACTER PRESENT.
D765D ; ENTRY: ND PARAMETERS
D766D ; EXIT: (A)=CHARACTER
D767D ; ALL REGISTERS SAEVO EXCEPT A
D768D ;

84F3 E5      D769D INPUTW PUSH HL ;SAVE REGISTERS
84F4 2165DD  D770D      LO      HL,DBOELP ;MINIMUM DELAY+1
84F7 C0CD85  D771D      CALL     OELAY ;AVDID PREVIOUS CHAR
84FA 2165DD  D772D      LD      HL,DBDELP ;MINIMUM DELAY + 1
84FD 22ED85  D773D      LO      (TSLC),HL ;INITIALIZE
85DD 21F592  D774D      LO      HL,IBUF ;START DF INPUT BUFFER
8503 22F285  D775D      LO      (IBUFN),HL ;RESET NEXT CHAR SLOT PNTR
85D6 22F885  D776D      LD      (IBUFL),HL ;RESET LAST CHARACTER SLDT
85D9 CD2984  D777D INWD10 CALL INPUT ;SCAN
85DC C5      D778D      PUSH     BC
85DD CD1585  D779D      CALL     GETCHR ;GET PDSSIBLE CHARACTER
851D C1      D78DD      PDP      BC
8511 28F6    D781D      JR      Z,INWD10 ;GD IF NDTHING
8513 E1      D782D      PDP      HL ;RESTDRE REGISTERS
8514 C9      D783D      RET      ;RETURN
D784D ;
D785D ;*****GET CHARACTER SUBRDUTINE*****
D786D ; GETS CHARACTER FRDM INPUT BUFFER IF THERE IS DNE
D787D ; ENTRY: ND PARAMETERS
D788D ; EXIT: (A)=ASCII CHARACTER DR ZEROES IF NDNE
D789D ; Z IF NDNE, NZ IF CHARACTER
D790D ; (B)=8DH IF SHIFT, D IF ND SHIFT
D791D ; ALL REGISTERS SAVED EXCEPT A,BC
D792D ;

8515 E5      D793D GETCHR PUSH HL ;SAVE REGISTERS
8516 2AFD85  D794D      LO      HL,(IBUFL) ;LAST CHARACTER
8519 E04BF285 D795D      LO      BC,(IBUFN) ;NEXT SLDT
851D B7      D796D      OR      A ;RESET CARRY
851E ED42    D797D      SBC      HL,BC ;TEST FOR END
852D 281E    D798D      JR      Z,GETD7D ;GD IF CAUGHT UP
8522 09      D799D      AOO      HL,BC ;RESTORE IBUFL
8523 7E      D80DD      LO      A,(HL) ;GET NEXT CHARACTER
8524 E68D    D8D1D      AND      8DH ;MASK IN SHIFT
8526 47      D8D2D      LO      B,A ;B NDW HAS SHIFT BIT
8527 7E      D803D      LO      A,(HL) ;NEXT CHARACTER
8528 E67F    D8D4D      AND      7FH ;A NDW HAS CHARACTER
852A C5      D8D5D      PUSH     BC ;SAVE SHIFT
852B F5      D8D6D      PUSH     AF ;SAVE CHARACTER
852C 23      D8D7D      INC      HL ;BUMP LAST PNTR
852D 22FD85  D8D8D      LD      (IBUFL),HL ;SAVE
853D D1F593  D8D9D      LD      BC,IBUFE ;END DF I BUF
8533 B7      D81DD      DR      A ;RESET CARRY
8534 E042    D811D      SBC      HL,BC ;TEST FDR ENO
8536 2D06    D812D      JR      NZ,GETD65 ;GD IF NDT END
8538 21F592  D813D      LD      HL,IBUF ;START DF I BUF
853B 22F085  D814D      LO      (IBUFL),HL ;BACK TD BEGINNING
853E F1      D815D GET065 PDP AF ;RESTORE CHARACTER
853F C1      D816D      POP      BC ;RESTDRE SHIFT
854D E1      D817D GETD7D PDP HL ;RESTDRE ENTRY REGISTER
8541 C9      D818D      RET      ;RETURN
D819D ;
D820D ;*****RANOOM NUMBER ROUTINE*****
D821D ; GENERATES A PSEUDD-RANDDM # FRDM D TO 127.
D822D ; ENTRY: ND PARAMETERS
D823D ; EXIT: (BC)=RANDDM # D-127
D824D ; ALL REGISTERS SAVED EXCEPT BC.
D825D ;

8542 F5      D826D RANO PUSH AF ;SAVE REGISTERS
8543 05      D827D      PUSH     OE
8544 E5      D828D      PUSH     HL
8545 E05BE185 D829D      LD      DE,(SEE0) ;GET SEED
8549 2AE385  D830D      LO      HL,(SEE0+2)
854C D67      D831D      LD      B,7 ;CDUNT FOR MULTIPLY BY 128
854E CD6B85  D832D RMD1D CALL SHIFT ;SHIFT DNE BIT LEFT
8551 1DFB    D833D      DJNZ     ROMD10 ;SEE0*128

```

```

8553 0603      08340      LO      B,3      ;FOR SUBTRACT
8555 C07185    08350      RDM020 CALL      SUB      ;SUBTRACT ONE
8558 10FB      08360      DJNZ     ROM020    ;SEEO*128-3*SEED=SEED*125
855A E053E185  08370      LD      (SEED),DE  ;STORE NEW SEED
855E 22E385    08380      LO      (SEED+2),HL
8561 3ETF      08390      LD      A,7FH      ;MASK
8563 A2        08400      AND      D          ;GET 0-127
8564 4F        08410      LD      C,A        ;NOW IN C
8565 0600      08420      LD      B,0        ;NOW IN BC
8567 E1        08430      POP      HL        ;RESTORE REGISTERS
8568 01        08440      POP      OE
8569 F1        08450      POP      AF
856A C9        08460      RET              ;SHIFT HL
8570 ;          08470 ;
8571 ;          08480 ;*****SHIFT SUBROUTINE*****
8572 ;          08490 ; SHIFTS CONTENTS OF (DE,HL) ONE BIT LEFT *
8573 ;          08500 ; ENTRY: NUMBER TO BE SHIFTEO IN (OE,HL) *
8574 ;          08510 ; EXIT: (DE,HL) SHIFTEO LEFT ONE BIT, LOGICAL *
8575 ;          08520 ; ALL REGISTERS SAVED EXCEPT OE,HL. *
8576 ;          08530 ;
8577 29        08540      SHIFT ADO      HL,HL      ;SHIFT HL
8578 EB        08550      EX      DE,HL      ;GET MS BYTE
8579 E06A      08560      AOC     HL,HL      ;SHIFT MS 2 BYTES AND CARRY
857A EB        08570      EX      OE,HL      ;NOW ORIGINAL*2
857B C9        08580      RET              ;RETURN
857C ;          08590 ;
857D ;          08600 ;*****SUBTRACT SEEO SUBROUTINE*****
857E ;          08610 ; SUBTRACTS FOUR BYTES OF SEEO FROM (OE,HL). *
857F ;          08620 ; ENTRY: (SEED - SEED+3)=SEED # *
8580 ;          08630 ; (OE,HL)=FOUR-BYTE VALUE *
8581 ;          08640 ; EXIT: (DE,HL)=RESULT OF SUBTRACT *
8582 ;          08650 ; ALL REGISTERS SAVED EXCEPT OE,HL *
8583 ;          08660 ;
8584 C5        08670      SUB      PUSH     BC      ;SAVE REGISTERS
8585 ED4BE385   08680      LD      BC,(SEED+2) ;GET LS BYTE
8586 B7        08690      OR      A          ;RESET CARRY
8587 ED42      08700      SBC     HL,BC      ;SUBTRACT LS 2 BYTES
8588 EB        08710      EX      DE,HL      ;GET MS 2 BYTES
8589 ED4BE185  08720      LD      BC,(SEED)   ;GET MS 2 BYTES
858A EB        08730      SBC     HL,BC      ;SUBTRACT MS 2 BYTES AND CY
858B EB        08740      EX      OE,HL      ;NOW ORIGINAL-SEED
858C C1        08750      POP      BC        ;RESTORE REGISTERS
858D C9        08760      RET              ;RETURN
858E ;          08770 ;
858F ;          08780 ;*****FILL CHARACTER SUBROUTINE*****
8590 ;          08790 ; FILLS DESIGNATED AREA WITH GIVEN CHARACTER *
8591 ;          08800 ; ENTRY: (A)=CHARACTER *
8592 ;          08810 ; (OE)=AREA *
8593 ;          08820 ; (BC)=NUMBER OF BYTES, 1-65525; 0 IS 65536 *
8594 ;          08830 ; ALL REGISTERS SAVED EXCEPT BC,DE *
8595 ;          08840 ;
8596 12        08850      FILLCH LD      (DE),A      ;FILL CHARACTER
8597 13        08860      INC     OE          ;BUMP POINTER
8598 0B        08870      DEC     BC          ;DECREMENT COUNT
8599 F5        08880      PUSH    AF          ;SAVE FILL CHAR
859A 78        08890      LO      A,B        ;TEST FOR ZERO
859B B1        08900      OR      C          ;
859C 2803      08910      JR      Z,FIL010    ;GO IF DONE
859D B1        08920      POP      AF          ;RESTORE FILL CHAR
859E 18F5      08930      JR      FILLCH     ;CONTINUE
859F F1        08940      FIL010 POP      AF          ;RESTORE A
859A C9        08950      RET              ;RETURN
859B ;          08960 ;
859C ;          08970 ;*****DECIMAL TO BINARY CONVERSION SUBROUTINE*****
859D ;          08980 ; CONVERTS UP TO SIX ASCII CHARACTERS REPRESENTING *
859E ;          08990 ; DECIMAL NUMBER TO BINARY. MAXIMUM VALUE IS 65535. *
859F ;          09000 ; ENTRY: (HL)=BUFFER CONTAINING ASCII *
859A ;          09010 ; (B)=NUMBER OF CHARACTERS *
859B ;          09020 ; EXIT: (HL)=BINARY # 0-65535 *
859C ;          09030 ; NZ IF INVALID ASCII CHARACTER OTHERWISE Z *
859D ;          09040 ; ALL REGISTERS SAVED EXCEPT A,HL *
859E ;          09050 ;
859F C5        09060      DECBIN PUSH     BC      ;SAVE REGISTERS
859A 05        09070      PUSH    OE          ;
859B 00E5      09080      PUSH    IX          ;
859C 0D210000  09090      LD      IX,0        ;SET RESULT
859D DD29      09100      DEC040 ADD      IX,IX  ;INTERMEDIATE*2
859A DE5       09110      PUSH    IX          ;
859C 0029      09120      ADO     IX,IX      ;*4

```



```

859E DD29      09130      ADD      IX,IX      ;#8
85A0 D1        09140      POP      DE          ;#2
85A1 DD19      09150      ADD      IX,DE      ;#10
85A3 7E        09160      LD      A,(HL)      ;GET CHARACTER
85A4 D630      09170      SUB      30H        ;CONVERT
85A6 FAB685    09180      JP      M,DEC070    ;GO IF LT "0"
85A9 FE0A      09190      CP      10         ;TEST FOR GT "9"
85AB F2B685    09200      JP      P,DEC070    ;GO IF GT "9"
85AE 5F        09210      LD      E,A        ;NOW IN E
85AF 1600      09220      LD      D,0        ;NOW IN DE
85B1 DD19      09230      ADD      IX,DE      ;MERGE
85B3 23        09240      INC      HL
85B4 10E2      09250      DJNZ     DEC040     ;GO IF MORE
85B6 78        09260      LD      A,B        ;COUNT TO A
85B7 B7        09270      OR      A          ;SET OR RESET Z FLAG
85B8 DDE5      09280      PUSH     IX        ;RESULT TO HL
85BA E1        09290      POP      HL
85BB DDE1      09300      POP      IX        ;RESTORE REGISTERS
85BD D1        09310      POP      DE
85BE C1        09320      POP      BC
85BF C9        09330      RET              ;RETURN
09340 ;
09350 ;*****DELAY SUBROUTINE*****
09360 ; DELAYS 1 TO 65536 MILLISECONDS.
09370 ; ENTRY: (HL)=DELAY COUNT IN MILLISECONDS
09380 ; 0=65536
09390 ; ALL REGISTERS SAVED.
09400 ;
85C0 C5        09410      DELAY     PUSH     BC          ;SAVE REGISTERS
85C1 D5        09420      PUSH     DE
85C2 E5        09430      PUSH     HL
85C3 11FFFF    09440      LD      DE,-1      ;-1 FOR DECREMENT
85C6 2B        09450      DEC      HL        ;ADJUST FOR "JP NC"
85C7 0682      09460      DEL010   LD      B,130    ;INNER LOOP TIMING
85C9 10FE      09470      DEL020   DJNZ     DEL020    ;LOOP FOR 1 MLLISEC
85CB E5        09480      PUSH     HL        ;SAVE COUNT
85CC 2AED85    09490      LD      HL,(TSLC)   ;GET TIMER COUNT
85CF 23        09500      INC      HL        ;BUMP
85D0 22ED85    09510      LD      (TSLC),HL ;SAVE
85D3 E1        09520      POP      HL        ;RESTORE OUTER LOOP CNT
85D4 19        09530      ADD      HL,DE      ;DECREMENT OUTER LOOP CNT
85D5 DAC785    09540      JP      C,DEL010   ;CONTINUE
85D8 E1        09550      POP      HL        ;RESTORE REGISTERS
85D9 D1        09560      POP      DE
85DA C1        09570      POP      BC
85DB C9        09580      RET              ;RETURN
09590 ;
09600 ;*****WORKING STORAGE*****
09610 ;
85DC 0000      09620      TMP1     DEFW     0          ;TEMPORARY STORAGE
85DE 00        09630      PRINTF   DEFB     0          ;PRINTER FLAG:0=OFF,1=ON
85DF 00        09640      LPFTF    DEFB     0          ;LP 1ST TIME FLAG:0=1ST TIME
85E0 00        09650      CHARCT    DEFB     0          ;LP CHARACTER COUNTER
85E1 D204      09660      SEED      DEFW     1234      ;DEFAULT SEED
85E3 2E16      09670      DEFW     5678
85E5 9001      09680      DOTO      DEFW     400      ;DOT ON TIME (3 WPM DEFAULT)
85E7 B004      09690      DASHO     DEFW     1200     ;DASH ON TIME (3 WPM DEFLT)
85E9 003C      09700      CURCUR    DEFW     3C00H    ;CURRENT CURSOR POSITION
85EB 003C      09710      LSTCUR    DEFW     3C00H    ;LAST CURSOR POSITION
85ED 0000      09720      TSLC      DEFW     0          ;TIME IN MS SINCE LAST CHAR
85EF 20        09730      LASTR      DEFB     ' '      ;LAST RANDOM CHARACTER SENT
85F0 F592      09740      IBUFL      DEFW     IBUF     ;POINTER TO LAST IBUF SLOT
85F2 F592      09750      IBUFN      DEFW     IBUF     ;POINTER TO NEXT IBUF SLOT
09760 ;
09770 ;*****SYSTEM MESSAGES*****
09780 ;
85F4 20        09790      MSG1      DEFM     '          ***MORG***
20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20 20
20 20 20 2A 2A 2A 4D 4F
52 47 2A 2A 2A 20 20 20
20 20 20 20 20 20 20 20
20 20 20 20 20 20 20 20
20 20 20 20 20
8634 43        09800      DEFM     'CHAR=SEND CHARACTER SHIFT 0-9=SEND MSG N'
48 41 52 3D 53 45 4E 44
20 43 48 41 52 41 43 54
45 52 20 20 53 48 49 46

```

```

54 20 30 2D 39 3D 53 45
4E 44 20 4D 53 47 20 4E

865D 20      09810      DEFM      '  SHIFT R=SEND RANDOM  SHIFT D=DEFINE MS'
20 53 48 49 46 54 20 52
3D 53 45 4E 44 20 52 41
4E 44 4F 4D 20 20 53 48
49 46 54 20 44 3D 44 45
46 49 4E 45 20 4D 53

8685 47      09820      DEFM      'G  SHIFT S=DEFINE SPEED  SHIFT P,N=PRINT'
20 20 20 53 48 49 46 54
20 53 3D 44 45 46 49 4E
45 20 53 50 45 45 44 20
20 53 48 49 46 54 20 50
2C 4E 3D 50 52 49 4E 54

86AE 20      09830      DEFM      ' OR NO'
4F 52 20 4E 4F

86B4 00      09840      DEFB      0
86B5 50      09850      MSG2      DEFM      'PRINT SET'
52 49 4E 54 20 53 45 54

86BE 00      09860      DEFB      0
86BF 50      09870      MSG3      DEFM      'PRINT RESET'
52 49 4E 54 20 52 45 53
45 54

86CA 00      09880      DEFB      0
86CB 53      09890      MSG4      DEFM      'SET SPEED MODE. ENTER SPEED 3 TO 60 WPM: '
45 54 20 53 50 45 45 44
20 4D 4F 44 45 2E 20 45
4E 54 45 52 20 53 50 45
45 44 20 33 20 54 4F 20
36 30 20 57 50 4D 3A 20

86F4 00      09900      DEFB      0
86F5 44      09910      MSG5      DEFM      'DEFINE MESSAGE MODE. ENTER MESSAGE # 0-9: '
45 46 49 4E 45 20 4D 45
53 53 41 47 45 20 4D 4F
44 45 2E 20 45 4E 54 45
52 20 4D 45 53 53 41 47
45 20 23 20 30 2D 39 3A
20

871F 00      09920      DEFB      0
8720 49      09930      MSG6      DEFM      'INVALID SPEED. MUST BE 3 TO 60'
4E 56 41 4C 49 44 20 53
50 45 45 44 2E 20 4D 55
53 54 20 42 45 20 33 20
54 4F 20 36 30

873E 00      09940      DEFB      0
873F 52      09950      MSG7      DEFM      'RANDOM CHARACTER MODE. PRESS CLEAR TO STOP'
41 4E 44 4F 4D 20 43 48
41 52 41 43 54 45 52 20
4D 4F 44 45 2E 20 50 52
45 53 53 20 43 4C 45 41
52 20 54 4F 20 53 54 4F
50

8769 00      09960      DEFB      0
876A 54      09970      MSG8      DEFM      'TRANSMIT MESSAGE MODE. PRESS CLEAR TO STOP'
52 41 4E 53 4D 49 54 20
4D 45 53 53 41 47 45 20
4D 4F 44 45 2E 20 50 52
45 53 53 20 43 4C 45 41
52 20 54 4F 20 53 54 4F
50

8794 00      09980      DEFB      0
8795 4E      09990      MSG9      DEFM      'NO MESSAGE BY THAT #'
4F 20 4D 45 53 53 41 47
45 20 42 59 20 54 48 41
54 20 23

87A9 00      10000      DEFB      0
87AA 49      10010      MSG10     DEFM      'INVALID MESSAGE #. MUST BE 0-9'
4E 56 41 4C 49 44 20 4D
45 53 53 41 47 45 20 23
2E 20 4D 55 53 54 20 42
45 20 30 2D 39

87C8 00      10020      DEFB      0
87C9 45      10030      MSG11     DEFM      'ENTER MESSAGE. TERMINATE BY ENTER'
4E 54 45 52 20 4D 45 53
53 41 47 45 2E 20 54 45

```

```

52 4D 49 4E 41 54 45 20
42 59 20 45 4E 54 45 52

87EA 00      10040      DEFB      0
87EB 40      10050 MSG12  DEFM      'MORE THAN 256 CHARACTERS. 256 ACCEPTED'
      4F 52 45      20 54 48 41 4E
      20 32 35      36 20 43 48 41
      52 41 43      54 45 52 53 2E
      20 32 35      36 20 41 43 43
      45 50 54      45 44

8811 00      10060      DEFB      0
      10070 ;
      10080 ;*****CTAB CHARACTER TABLE*****
      10090 ;*   TABLE OF CHARACTERS TO BE SENT IN RANOOM MOOE.   *
      10100 ;*   DISTRIBUTION DOES NOT CORRESPOND TO THAT IN NOR-  *
      10110 ;*   MAL TEXT. SPACE CHARACTER NOMINALLY EVERY 5TH  *
      10120 ;*   CHARACTER.                                         *
      10130 ;
      10140 CTAB      OEFM      'ABCOEFGHIJKLMNOPQRSTUVWXYZ0123456789.,?/- =;'

8812 41      10150      DEFM      '0123456789.,?/- =:ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      42 43 44      45 46 47 48 49
      4A 4B 4C      40 4E 4F 50 51
      52 53 54      55 56 57 58 59
      5A 30 31      32 33 34 35 36
      37 38 39      2E 2C 3F 2F 2D
      20 30 3B

883E 30      10150      DEFM      '0123456789.,?/- =:ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      31 32 33 34 35 36 37 38
      39 2E 2C 3F 2F 2D 20 3D
      3B 41 42 43 44 45 46 47
      48 49 4A 4B 4C 4D 4E 4F
      50 51 52 53 54 55 56 57
      58 59 5A

886A 20      10160      DEFM      '
      20 20 20 20 20 20 20 20
      20 20 20 20 20 20 20 20
      20

887C 41      10170      OEFM      'ABCOEFGHIJKLMNOPRSTUVY'
      42 43 44      45 46 47 48 49
      4A 4B 4C      40 4E 4F 50 52
      53 54 55      56 59
      10180 ;
      10190 ;*****CDTAB CODE TABLE*****
      10200 ;*   TABLE OF VALIO ASCII CHARACTERS TO BE TRANSMITTED. *
      10210 ;*   INOEX TO CHARACTER USEO TO OBTAIN TIMING CODES   *
      10220 ;*   FROM TTAB.                                         *
      10230 ;
      10240 CDTAB      EQU      CTAB      ;SAME OATA
      10250 COTABS      EQU      44      ;SIZE OF DATA
      10260 ;
      10270 ;*****TTAB TIMING TABLE*****
      10280 ;*   TABLE OF TIMING COOES FOR CHARACTERS IN CDTAB.   *
      10290 ;*   POSITION IN THIS TABLE CORRESPONOS TO POSITION IN *
      10300 ;*   COTAB. EACH ENTRY IS 2 BYTES LONG.               *
      10310 ;*****
      10320 ;
      10330 TTAB      OEFW      34H      ;A
      10340      OEFW      301H      ;B
      10350      OEFW      311H      ;C
      10360      OEFW      0C1H      ;D
      10370      OEFW      0CH      ;E
      10380      DEFW      310H      ;F
      10390      OEFW      0C5H      ;G
      10400      DEFW      300H      ;H
      10410      OEFW      30H      ;I
      10420      DEFW      354H      ;J
      10430      OEFW      0D1H      ;K
      10440      OEFW      304H      ;L
      10450      DEFW      35H      ;M
      10460      OEFW      31H      ;N
      10470      OEFW      0D5H      ;O
      10480      OEFW      314H      ;P
      10490      OEFW      345H      ;Q
      10500      DEFW      0C4H      ;R
      10510      OEFW      0C0H      ;S
      10520      DEFW      0DH      ;T
      10530      OEFW      000H      ;U
      10540      OEFW      340H      ;V
      10550      OEFW      004H      ;W
      10560      DEFW      341H      ;X

8892 3400    10330 TTAB      OEFW      34H      ;A
8894 0103    10340      OEFW      301H      ;B
8896 1103    10350      OEFW      311H      ;C
8898 C100    10360      OEFW      0C1H      ;D
889A 0C00    10370      OEFW      0CH      ;E
889C 1003    10380      DEFW      310H      ;F
889E C500    10390      OEFW      0C5H      ;G
88A0 0003    10400      DEFW      300H      ;H
88A2 3000    10410      OEFW      30H      ;I
88A4 5403    10420      DEFW      354H      ;J
88A6 D100    10430      OEFW      0D1H      ;K
88A8 0403    10440      OEFW      304H      ;L
88AA 3500    10450      DEFW      35H      ;M
88AC 3100    10460      OEFW      31H      ;N
88AE 0500    10470      OEFW      0D5H      ;O
88B0 1403    10480      OEFW      314H      ;P
88B2 4503    10490      OEFW      345H      ;Q
88B4 C400    10500      DEFW      0C4H      ;R
88B6 C000    10510      OEFW      0C0H      ;S
88B8 0D00    10520      DEFW      0DH      ;T
88BA 0000    10530      OEFW      000H      ;U
88BC 4003    10540      OEFW      340H      ;V
88BE 0400    10550      OEFW      004H      ;W
88C0 4103    10560      DEFW      341H      ;X

```

```

88C2 5103      10570      0EFW      351H      ;Y
88C4 0503      10580      0EFW      305H      ;Z
88C6 5500      10590      0EFW      0055H     ;0
88C8 5400      10600      0EFW      0054H     ;1
88CA 5000      10610      0EFW      0050H     ;2
88CC 4000      10620      0EFW      0040H     ;3
88CE 0000      10630      0EFW      0000H     ;4
8800 000C      10640      0EFW      0C00H     ;5
8802 010C      10650      0EFW      0C01H     ;6
8804 050C      10660      0EFW      0C05H     ;7
8806 150C      10670      0EFW      0C15H     ;8
8808 550C      10680      0EFW      0C55H     ;9
880A 4434      10690      0EFW      3444H     ;.
880C 0535      10700      0EFW      3505H     ;,
88DE 5030      10710      0EFW      3050H     ;?
88E0 410C      10720      0EFW      0C41H     ;/
88E2 0134      10730      0EFW      3401H     ;-(.....)
88E4 AA03      10740      0EFW      3AAH      ;BLANK
88E6 440C      10750      0EFW      0C44H     ;(-.-.-)
88E8 0000      10760      0EFW      0          ;ERROR(.....)
10770 ;
10780 ;*****MESSAGE BUFFER*****
10790 ;*   ARBITRARILY SET AT 2560 BYTES (256 BYTES PER MSG   *
10800 ;*   PLUS MSG# PLUS 1 TERMINATOR).                      *
10810 ;
88EA          10820 MBUF      EQU      $
92F5          10830 ENOM      EQU      $+2571
10840 ;
10850 ;*****INPUT BUFFER*****
10860 ;*   CIRCULAR INPUT BUFFER OF 256 BYTES                *
10870 ;
92F5          10880 IBUF      EQU      ENOM
93F5          10890 IBUFE     EQU      IBUF+256
9459          10900 TOPS      EQU      IBUFE+100      ;TOP OF STACK
8000          10910 ENO       START
00000 TOTAL ERRORS

```

```

BTAB      80C0      COTAB      8812      COTABS      002C      CHARCT      85E0      CLEAR      0001
CLRCOM     83FB      COFF       8356      CON         8330      CTAB       8812      CURCUR     85E9
OASH       8335      OASH0      85E7      OBOEL        0064      OBOELP     0065      OEC040     8598
OEC070     85B6      OECBIN     8590      0EF005       80F1      0EF025     8126      0EF030     812F
0EF035     8140      0EF040     8152      0EF050       817F      0EFINE     80EB      0ELO10     85C7
0ELO20     85C9      0ELAY      85C0      OIS010       8303      OISCHR     83BE      00T        8327
00T0       85E5      OSP005     8370      OSP010       8370      OSPMES     8360      ENOM       92F5
ENTER      0002      FIL010     858E      FILLCH       8583      FN0020     8204      FNOMSG     82C7
FNOSR      8207      FTAB       80BE      FTABS        000F      GET065     853E      GET070     8540
GETCHR     8515      IBUF       92F5      IBUFE        93F5      IBUFL      85F0      IBUFN      85F2
INP010     8441      INP020     844C      INP025       844E      INP035     8456      INP040     8458
INP052     8478      INP055     847E      INP056       8497      INP057     849A      INP059     84B1
INP060     84B7      INP065     84B9      INPUT        8429      INPUTS     8400      INPUTW     84F3
INS010     8414      INS030     8426      INW010       8509      KBTAB      84BB      LASTR      85EF
LINE1      3C40      LINE11     3EC0      LINE12       3F00      LINE13     3F40      LINE14     3F80
LINE15     3FC0      LPFTF      850F      LPRO10       8398      LPRO20     83A4      LPRO90     83AC
LPRINT     8381      LPS010     83B0      LPSTAT       83AF      LSTCUR     85EB      MBUF       88EA
MLOEL      000A      MOR015     8037      MOR020       8064      MOR021     8066      MOR022     8070
MOR025     8083      MOR027     8096      MOR030       80AB      MSG1       85F4      MSG10      87AA
MSG11      87C9      MSG12      87EB      MSG2         86B5      MSG3       86BF      MSG4       86CB
MSG5       86F5      MSG6       8720      MSG7         873F      MSG8       876A      MSG9       8795
NOPRNT     82B1      OFF010     835E      ON010        8343      PRI010     829C      PRINT      82B8
PRINTF     850E      RAN010     8212      RAN020       8226      RAN0       8542      RANOOM     81F0
ROM010     854E      ROM020     8555      SCREEN       3C00      SCROLL     8306      SEO        85E1
SHIFT      856B      SNO010     82FE      SNOCHR       82E0      SPACE      832D      SPE005     8197
SPE015     81C3      SPE020     810F      SPEE0        8191      SPEE0F     03C0      START      8000
SUB         8571      TMP1       850C      TOPS         9459      TSLC       85E0      TTAB       8892
XMIT       8237      XMT010     825E      XMT020       8273

```

Figure 13-8. MORG Listing

The variables are followed by system messages and each message is terminated by a zero.

The CTAB table (Character Table) is a table of 128 ASCII characters distributed among all permissible characters. This table is used in the Random mode to generate a random character. A pseudo-random number of 0 through 127 is used as an index to pick up the corresponding character from the table.

The CDTAB (CoDe Table) is a table of 44 characters that represent all valid characters generated by the system. Since these are contained in the first 44 characters of CTAB, CDTAB is equated to CTAB.

TTAB (Timing Table) is a table of 44 entries associated with CDTAB. Each entry is two bytes (one word) long and represents the coded eight-field representation of the Morse code character. CDTAB and TTAB are used to convert a valid character to its dot and dash equivalent.

The next object is MBUF (Memory Buffer), a 2571-byte table that holds all defined messages. Since each message can consist of 256 characters plus a one-byte header of the message number, MBUF is 257×10 bytes long plus one byte. The last byte ensures that there'll always be a -1 terminator for a full buffer.

IBUF (Input Buffer) is a circular buffer that holds the current characters read from the keyboard. You can use it to implement the **buffered** input while characters are being generated to the cassette output. TOPS is the "top of stack," 101 bytes up from the end of the IBUF area.

The memory map for MORG is shown in Figure 13-9.

Program Description

We'll use a "bottom-up" approach to describe the program modules. We'll start at the lowest-level modules and work our way up to a description of how the upper two levels utilize the other modules to implement the Morse code program functions.

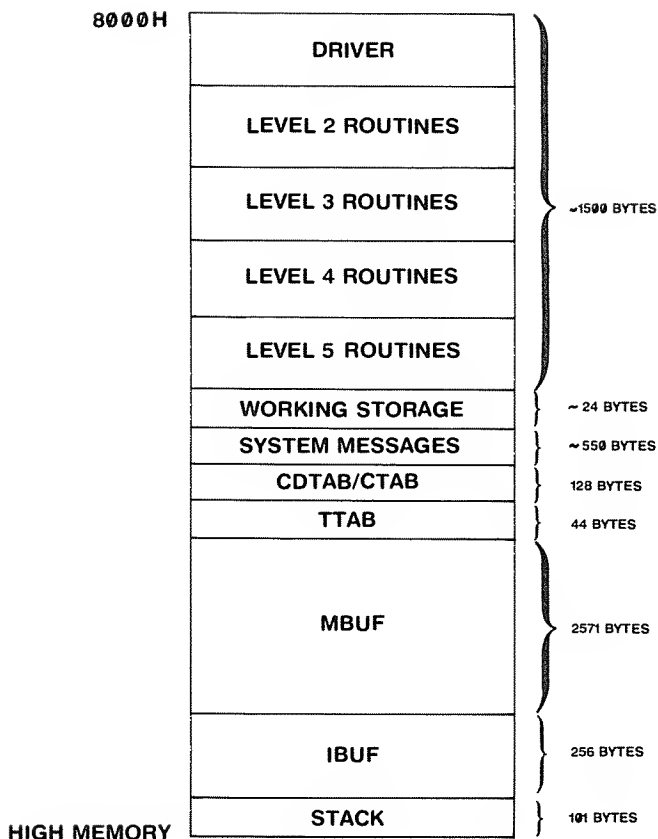


Figure 13-9. MORG Memory Map

Delay Subroutine (DELAY)

The DELAY subroutine delays from 1 to 65536 milliseconds, depending upon the input count in the HL register pair. In addition to the delay, it increments the software “elapsed time” variable, TSLC, by the timing count. TSLC holds the rough elapsed time in milliseconds and is used only for timing the debounce delay.

DELAY works by decrementing the delay count in HL by -1 in DE. Note that the decrement is done by an ADD HL,DE, which produces a carry if the contents of HL are positive or zero.

Decimal-to-Binary Conversion Subroutine (DECBIN)

This subroutine takes a string of ASCII characters representing decimal digits and converts it into a binary number of 0 through 65535. Entry is made with HL pointing to the buffer containing the leftmost character of the string and B containing the number of characters in the string.

The result is in HL on exit. If an invalid character is found, DECBIN stops conversion and returns with the Z flag reset (NZ). An invalid character is defined as one that's not 30H(0) through 39H(9).

DECBIN performs the conversion by taking an ASCII character, subtracting 30H to change it into binary 0 through 9, testing the result for validity, and adding the 0 through 9 to a partial result in IX.

The loop at DEC040 is entered once for each character of the string. Each time the loop is entered, the partial result is multiplied by 10 with a shift-and-add technique.

Fill Character Subroutine (FILLCH)

The Fill Character subroutine fills a specified area with a given fill character. It operates with a simple loop that tests for a decrement of BC down to zero.

Subtract Seed Subroutine (SUB)

The Subtract Seed subroutine is a four-byte multiple-precision subtract that subtracts the contents of DE,HL, (treated as a four-byte integer value) from the "SEED" variable, a four-byte value in SEED through SEED+3. The result goes back to SEED. Two subtracts are done, the second using a possible carry from the lower-order subtract.

Shift Subroutine (SHIFT)

The SHIFT subroutine shifts the contents of DE,HL, treated as a four-byte integer, one bit position left in logical fashion. SHIFT is used to multiply by two and is called by the RAND routine.

Random Number Subroutine (RAND)

RAND is a subroutine to generate a pseudo-random number. The algorithm for the routine is this: Starting with a "seed" value, a new pseudo-random number is generated by multiplying by an odd power of 5, **modulus** 64K. "Modulus" simply means that the result is divided by 64K, the quotient is discarded, and the remainder is saved. This process is done automatically as the result is held in a four-byte integer variable in SEED through SEED+3; the maximum value here is 65535.

The odd power of five chosen here is 125 (5 to the third power). The old "seed" is put into DE,HL from SEED. The SHIFT subroutine is then called 7 times to multiply the value by 128. The original value in SEED is then subtracted three times from DE,HL by calling the SUB subroutine three times. The result is the old seed*125. This value is put back into variable SEED for the next generation.

Hints and Kinks 13-5

Random Number Notes

The algorithm used here is

$$R(N+1) = 125 * R(N) \bmod 2^{32}$$

The mod 2^{32} operation is automatically performed by working in 32 bits.

You can generate about 2^{30} or 1 billion numbers without repeats in this approach. I haven't verified this, having quit after scanning only about 1 million cases.

A 7-bit **mask** of the high-order byte of the new random number is performed to get a value of 0 through 127. This is returned in BC. Of course, B will always be 0 on return, but it's convenient to have the number in a register pair for further processing.

Get Character Subroutine (GETCHR)

The Get Character subroutine gets the next character, if any, from the circular input buffer IBUF. During normal operation IBUF is being filled with characters as they are read in from the keyboard. The characters keep filling up IBUF as fast as they're input. Also during normal operation, characters are being output from IBUF to the Morse Code Send Character (SNDCHR) routine, and for display on the screen.

We use this routine to handle the task of seeing if another character is ready for output and display. The routine uses two pointers, IBUFL and IBUFN. IBUFL holds the address of the next character to be output. IBUFN holds the address of the next "slot" in IBUF. If these two pointers are equal, the GETCHR subroutine has caught up with the character storage.

If the pointers are not equal, GETCHR uses IBUFL to get the next character. It then bumps IBUFL by one to point to a possible next character. A **limit** condition occurs when IBUFL points to the last character position of IBUF+1. If this is true IBUFL is reset to the start of IBUF.

On exit, A contains the ASCII character found in IBUF or zeroes if there was no character available. The Z flag is either set if there was no character or reset if there was a character.

Input Character and Wait (INPUTW)

The Input Character and Wait subroutine is called to pick up a single character for user input. It is not used during normal operation, only for response to system queries such as code speed and message number. In these cases, no buffering need be used.

INPUTW calls INPUT to input the next character. INPUT is the **fast** input routine used for all character input. Before calling INPUT, INPUTW “dummies up” the “Time Since Last Character” variable TSCL to make it look like the debounce time has elapsed. It does this each time a new character is required. This approach makes it possible to use INPUT for both “wait until next character” and “scan fast and return” functions. INPUTW does not exit until a character has been input to the keyboard.

Keyboard Input Subroutine (INPUT)

This subroutine is the high-speed keyboard input subroutine that permits **buffering** of input characters during transmission of code characters. INPUT has three basic functions: fast scan, conversion, and storage.

When INPUT is called it loads the A register with input/output address 387FH. This address enables all “row” addresses of the keyboard. If you are pressing **any** key, there will be a one bit in A after this LD. In effect, this is an OR of all rows into eight column bits. If you aren’t pressing any key, INPUT immediately returns. This sequence takes six instructions.

If there is a one bit in A after the fast scan, INPUT checks the elapsed time by testing variable TSLC. If the debounce time has not elapsed (DBDEL), INPUT returns since the key press may represent the same key input on the last read.

If the debounce time has elapsed, INPUT goes into the conversion processing. It now scans each row by reading in I/O address 3801H, 02H, If a one bit is found after any read, the conversion routine at INP020 is entered. At this point HL contains the row address. The column bit is converted to a number from 0 through 7 and added to 8 times the row value to get an index of 0 through 56.

The index value is added to the address of the start of KBTAB. KBTAB is a table of 56 characters representing the keyboard configuration. KBTAB represents the “unshifted characters” from the keyboard. The character represented

by the keypress is read from KBTAB and put into A. If this character is a zero, it is an "ignore" character and a return is made.

The SHIFT key is now read and stored in B. (NOTE: On Model III, this is the LEFT SHIFT key.) If the character read is either slash (/) or minus (-), the SHIFT key is tested; if SHIFT is being pressed, the slash or minus is converted to a question mark (?) or equals (=) by setting bit 4 of the character.

A test is then made for the CLEAR character. The CLEAR key can be pressed to reset operation of the program. If CLEAR is being pressed, the program is restarted at MOR015 after some cleanup of the current cursor position (CURCUR).

If CLEAR is not being pressed, the code at INP057 is entered. This code stores the character in IBUF by using IBUFN, the variable that points to the next IBUF character "slot." After the store, IBUFN is incremented by one. A test is then made to see that IBUFN has not gone beyond the end of IBUF. If it has (IBUFN=IBUFE), IBUFN is reset to the beginning of IBUF. Since a character has just been read, the "time since last character" variable TSLC is reset to zero, and the subroutine is exited.

Input String Subroutine (INPUTS)

The Input String Subroutine is used to input a user response to a system question such as that for code speed. It calls INPUTW, Input Character and Wait. As each character is input, it's displayed on the screen by a CALL to DISCHR. INPUTS detects the end of the input string by testing for an ENTER and returns with a character count in B. The character count will be used in converting the (decimal) string to binary in DECBIN.

Clear Communications Area (CLRCOM)

The Clear Communications subroutine calls FILLCH to clear the last three lines on the screen (the communications area). Blanks are filled.

Scroll Subroutine (SCROLL)

SCROLL is used to scroll up the first 12 lines of the screen. This is the "text" area used to display text as Morse code characters are entered and transmitted. Lines 1 through 11 are moved up into lines 0 through 10 by an LDIR instruction. The last line of the text area, line 11, is then filled with blanks by a call to FILLCH.

Display Character Subroutine (DISCHR)

The Display Character Subroutine stores a character on the screen at the current cursor position. It is used for all screen output, both in the communications and text area. The current cursor position is always held in variable CURCUR as a screen address. After the character is stored, CURCUR is incremented by one. If it now points to the last position of line 11, a CALL to SCROLL is made.

Line Printer Subroutine (LPRINT)

LPRINT is the line printer driver subroutine. If the PRINTF variable is non-zero, the "P" command has been given for simultaneous line printer output. In this case, the character is output to the system line printer. On the first output to the line printer (LPFTF=0) and after 32 characters have been output to the line printer, the subroutine automatically outputs a carriage return for a new line. This eliminates overprinting in some system line printers.

LPRINT uses an "internal" subroutine LPSTAT to read line printer status.

Line Printer Output

LPSTAT is a typical line printer driver for parallel line printers. The handshaking between the computer and line printer is probably the simplest of any peripheral. A line printer is 'ready' when it has finished printing a character and ready to accept the next. The inverse of ready is 'busy.' In an unbuffered line printer, the busy time duration in seconds is $1/N$, where N is the number of characters per second.

A status loop checks the busy status (LD A,(37E8H)) until the line printer is ready to accept the next character, at which time the CPU outputs the character (LD (37E8H),A) causing the line printer to again become busy. The busy flag is set automatically by the line printer electronics during printing and reset after printing.

Buffered line printers can accept characters during printing as long as the buffer does not become full.

Display Message at Location N (DSPMES)

DSPMES is called for system messages, such as those prompting the user to enter code speed. On entry, HL points to the start of a message string. This string is assumed to be terminated by a zero (null) character. DSPMES picks up a character at a time and stores it on the screen by using the contents of BC as a pointer. HL and BC are incremented after each store. When a null is detected, DSPMES returns. Variable CURCUR, Current Cursor, is adjusted for each store.

Send Character Subroutine (SNDCHR)

This subroutine is the heart of MORG as far as transmission of audio code characters. It's called with the A register containing the ASCII character to be transmitted.

SNDCHR first searches for the character in CDTAB, the character code table. As SNDCHR can only be called with a valid character, the search must be successful. When the character is found in CDTAB, its index of 0 through 43 is multiplied by 2. The index now can be used to pick up the corresponding TTAB entry, which contains the coding of dots, dashes, spaces, and terminator. The character is picked up from TTAB by adding the index to TTAB and loading the character into H and L.

The code starting at SND010 and ending at the next return is the major "output" loop of SNDCHR. The contents of HL are shifted two bits right. The two bits shifted out are saved in A and tested. They are either 00 (dot), 01 (dash), 10 (space), or 11 (terminator). If the bits are a terminator, the subroutine is exited.

If the bits represent a dot, the DOT subroutine is called; if a dash, DASH is called.

DOT and DASH call two other subroutines at a lower level, CON and COFF. CON produces an audio tone of 500 hertz by outputting alternate 01 and 10 to the two least significant bits of the cassette latch at address OFFH. DELAY is called to time out the 1 millisecond delay between an on and off condition. The output goes on for 1 millisecond, then off for one millisecond, producing a 500 hertz **square wave**.

COFF is similar to CON, except that it leaves the cassette latch off and performs only the delay.

DOT calls CON to produce the tone for one dot time (DOTO) and then calls COFF for one dot time. DASH calls CON to produce the tone for one dash-time (DASHO) and then calls COFF for one **dot** time.

If the two-bit code was a 10 in the main loop of SNDCHR, an inter-character space is called for. In this case, only COFF is called for two dot times, as one dot time has already been output for the last dot or dash. The total elapsed time is three dot times.

Find Message Subroutine (FNDMSG)

This subroutine is used to search the Memory Buffer MBUF for a given message number. It's possible that the message has not been defined.

A search is made by scanning for a given message number 0 through 9. This byte can only exist if a message has been stored in MBUF, as other bytes are ASCII characters, minus ones, or other digits.

If the message number is found, FNDMSG is exited with HL pointing to the message, otherwise HL points to the next available area for the message. A CPIR instruction is used for the search in a small subroutine called FNDSR.

Main Driver

The subroutines above are used by the main body of code to implement the functions of MORG. The main **driver** of MORG starts at START. All "restarts" (as, for example, a CLEAR) come back to MOR015.

The code from START to MOR015 initializes MBUF with minus ones, initializes the SEED, clears the screen, and draws the line separating the text area from the communications area. The restart code at MOR015 initializes the IBUF pointers and TSLC and outputs the initial message.

MOR020 is the start of the loop to output characters or messages. A call is made to INPUT to scan the keyboard and then to GETCHR to get a possible character. If no character is available, a "minor" count in D is incremented. When this count is equal to MLDEL, approximately one millisecond has been reached, and variable TSLC is bumped by one. This software counting is necessary as TSLC is used in the debounce delay in INPUT. If it were not done, the keyboard routine could be "locked up" waiting for the debounce delay.

If a character is present, it's either a character to be transmitted or a special function that uses the shift key and 0 through 9, T (Transmit Message), D (Define

Message), S (Define Speed), R (Random), P (Print), or N (No Print). If no shift is present, the character is displayed (DISCHR), printed (LPRINT), and transmitted (SNDCHR) and a return is made to MOR020.

If a special SHIFT key has been pressed, a table of allowable functions, FTAB, is searched for the key. If none is found, the keypress is ignored. If a match is found, the index from FTAB is used to pick up an address from **branch table** BTAB. The entries in this table match the entries for the character. A branch is then made to one of six routines that will implement the function required. Return will be made back to MOR015 at the completion of the function.

Function Routines

The Define Message Routine (DEFINE) is entered if SHIFT D is pressed. It calls CLRCOM to clear the communication area and then prompts the user to input a message number. A test is made of the input for a valid number and an error message is output if the number is not 0 through 9.

FNDMSG is called to check for a current message with that number. If a current message is found, it's deleted by moving up the remainder of MBUF on top of the existing message. The remainder of MBUF is then filled with minus one bytes.

The loop at DEF040 is then executed. This loop uses INPUTW to get a character, store it in MBUF, and display it by DISCHR. A check is made for messages exceeding 256 characters with an appropriate message output if this is the case. The main driver loop at MOR015 is reentered when an ENTER is input, ending the message.

The Set Speed Routine (SPEED) is entered if a SHIFT S is input. It clears the communication area and displays a prompt message for user speed input. INPUTS is used to get a character string, which is then converted to binary by DECBIN.

If the speed value is less than 3 or greater than 60, an error message is output, and the sequence is repeated. Otherwise, the speed value is used to divide system value SPEED (nominally 1200, but adjusted for system overhead) to obtain the dot time in milliseconds. This value is stored in DOTO. The dot time is multiplied by three to obtain the dash time. This value is stored in DASHO. The main driver is then reentered at MOR015.

The Transmit Random Characters Routine is entered when a SHIFT R is input. This routine clears the communications area. It then calls RAND to get a random number of 0 through 127. This value is then used to get a character from the 128-byte CTAB. If a blank character is picked up, a test is made to see if the last character was blank. If it was, a new character is generated to avoid sending two blanks in a row.

The character is then displayed (DISCHR), printed (LPRINT), and transmitted (SNDCHR). A test is then made for a CLEAR character by calling INPUT even though no input is taking place for characters. A CLEAR will restart at MOR015, otherwise RAN010 is executed.

The Transmit Message Routine (XMIT) is entered if a SHIFT 0 through 9 is entered. The communications area is cleared. At this point, the message number is still in the BC register pair. It is used to call FNDMSG to find the location of the message in MBUF. If the message number is not found, an error message is briefly output and the main driver is reentered at MOR015.

If the message is found, it is transmitted character by character until a non-ASCII character is detected (new message number header or minus one). DISCHR, LPRINT, and SNDCHR are called to display, print, and send each character.

The Print (PRINT) and No Print (NOPRNT) Routines are entered by SHIFT P and SHIFT N, respectively. The only action in each is to clear the communications area and display a short PRINT SET or PRINT RESET message. The main driver is then reentered at MOR015.

Using This Program

There are several alternatives in using the program. First of all, of course, you don't really have to use it at all except for reference. It illustrates many of the concepts we discussed earlier in the book.

If you care to actually run the program, you may key in the machine code by using Disk DEBUG or T-BUG. The program is assembled at 8000H and, of course, is non-relocatable except by reassembly. At first blush, this seems like a formidable task, as there are several thousand bytes. However, it can be done in slightly over an hour for a fast typist. **Checkpoint** occasionally by saving what you've done. Also, mind the locations! There is nothing worse than completing a large input of several thousand locations and finding out at the end that you've been one location off for 1500 bytes!

A third alternative is to key in the source. This is a formidable job, but does allow you to experiment with the program once you have captured the program on disk or tape.

Chapter Fourteen

Tic-Tac-Toe Learning Program

We're presenting another larger assembly-language project in this chapter — a program that plays tic-tac-toe. The difference between this and other programs that play the game, however, is that this one learns! It starts off playing all possibilities, but quickly learns which paths lead to winning games.

The program was assembled using the Disk Editor/Assembler, but can be easily modified to run on EDTASM by the simple format changes of adding colons after labels and using single arguments on data generation pseudo-ops.

General Specifications

I can sum up the general specs for the program in a paragraph or so: Write a program that plays tic-tac-toe with a human. The program always makes the first move. The program must start out by playing any possible sequence and must not use any predetermined logic to decide where to play, except for the obvious capabilities of being able to block a human's winning row, column, or diagonal and of trying to complete its own winning row, column, or diagonal.

As the program plays games, it should record the win, loss, and draw records of the games. More importantly, it should "learn" which paths produced winning games and somehow reinforce those paths. It should also learn which paths caused it to lose and avoid those sequences.

The program should, of course, draw a tic-tac-toe grid, put in Xs and Os, prompt the human, display the

win/loss/draw record, and do all the usual things in interacting with the user.

Operation

Here again, as with the Morse Code Generator program, we can define the operation of the program before actually designing it — we can define the external or outward operation of the system. Whether or not we can actually implement such a program is still a question at this point.

The program should draw a tic-tac-toe grid on the screen as shown in Figure 14-1. The computer's symbol will be an X and the human's an O. Initially, and at each new game, the title `TIC-TAC-TOE` will be displayed at the top of the screen.

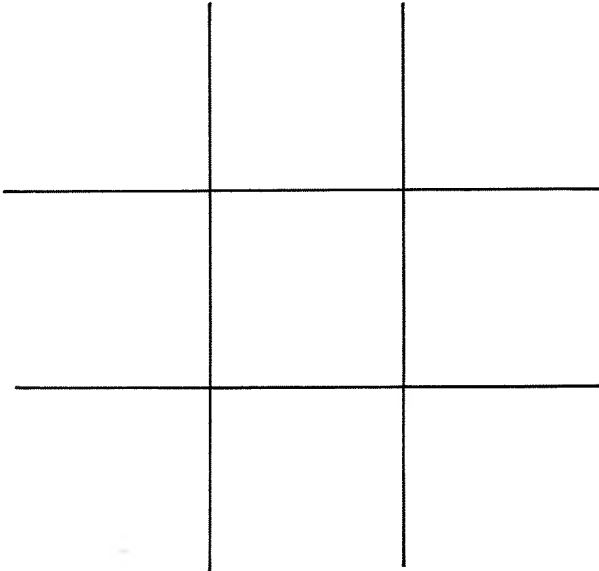


Figure 14-1. Tic-Tac-Toe Grid

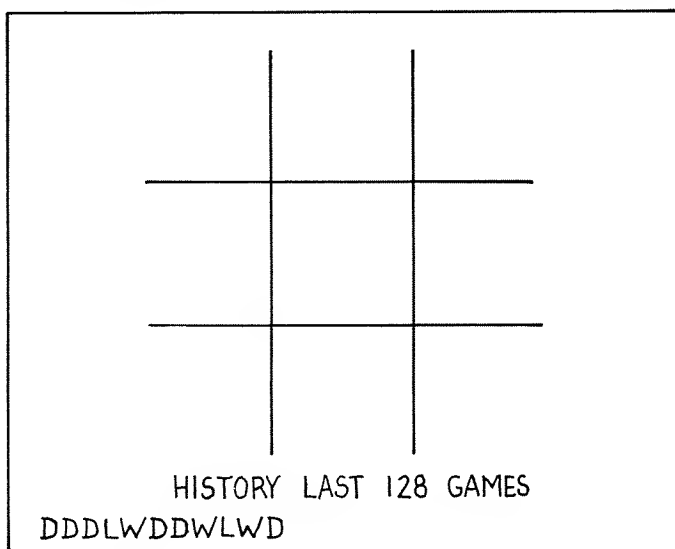
The computer moves first, placing an X in one of the nine positions, and then prompts the user with a `YOUR MOVE` message. The human then responds by entering a digit of 0 through 8, depending upon the position in which he wants to place an O. After the O is placed, the computer

plays again and prompts the user. This dialogue continues until either the human or the computer wins.

If the human makes an invalid choice of square (one that is already occupied by an X or O), the message TRY AGAIN is displayed, and the human must try a new move.

At the end of the game, the program displays either YOU WIN!, I WIN, or DRAW and then displays the message ONE MORE? If the human wants another game, he can press any key from 0 through 8 to start a new game.

At the end of each game, a "history" of the past 128 games is displayed on the bottom three lines of the screen as shown in Figure 14-2. There are 2 lines of 64 characters on the bottom of the screen. As each game is played, a W, L, or D is displayed in the next position for win, lose, or draw. When the 128 positions are filled up, the history "slides" to the left to display the last game and previous 127 games.



**Figure 14-2. Tic-Tac-Toe
History**

Messages are displayed in “large format” at the top of the screen, except for the history message, which is standard alphanumeric size.

All of the above seems perfectly achievable for an assembly-language or BASIC program, with the exception of the internal workings of the “learning” capability. We’ll discuss this major point next.

General Design

The general design of this program can be divided into several areas of research:

- Nature of tic-tac-toe
- Alternatives to learning
- Algorithms

Nature of Tic-Tac-Toe

Just to start from “square one,” let’s review the rules of tic-tac-toe. Tic-tac-toe is played on the grid shown in Figure 14-1. There are nine squares in the grid. Two opponents play, one using Xs and one using Os (an old name for the game is “naughts and crosses”).

Each player plays in turn, putting his X (or O) into a vacant square. The first player to successfully put all Xs or Os into a row, column, or diagonal wins the game. If neither player can complete a row, column, or diagonal, the game is a draw. A sample game is shown in Figure 14-3.

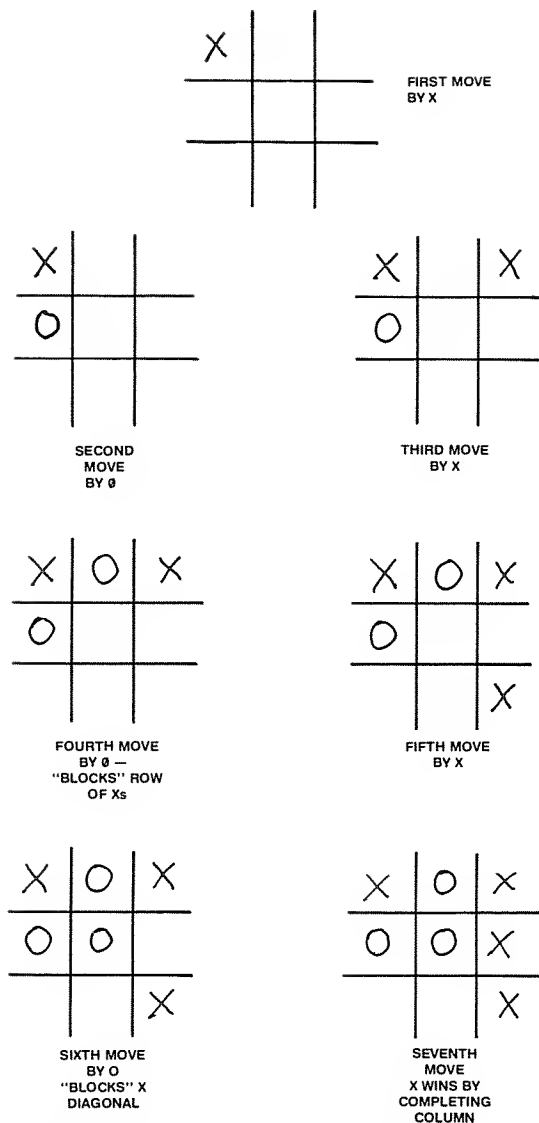


Figure 14-3. Sample Tic-Tac-Toe Game

Let's start our research by asking, "How many **different** games can be played?" At first, this seems like a question for a statistician, but we can draw some conclusions about it.

Each square can only have an X, O, or space (no play) in it. Therefore, we can represent any move of any game by listing all possible configurations of Xs, Os, and spaces. If we start to do this by hand, we can get confused quite easily, as shown in Figure 14-4.

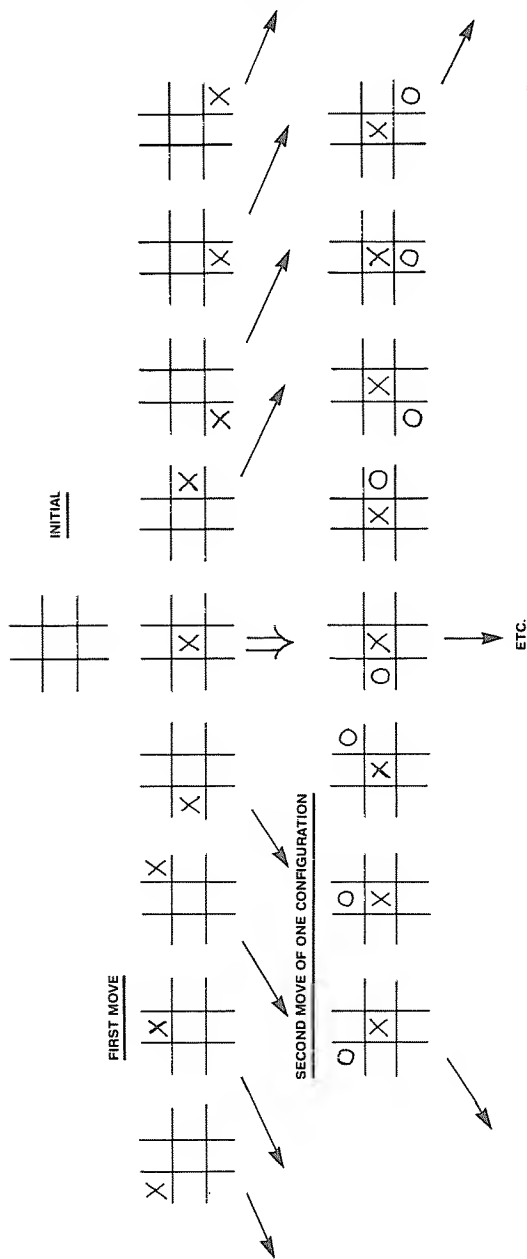


Figure 14-4. Manual Analysis of Tic-Tac-Toe

There are just too many combinations to keep track of! Let's try a different approach. We'll assign an X a one value, an O a two value, and a space a zero value. Now we can write down any configuration, beginning of game, middle of game, or end of game by a series of 0,1, and 2. We'll number the squares 0,1,2,3,4,5,6,7, and 8 as shown in Figure 14-5.

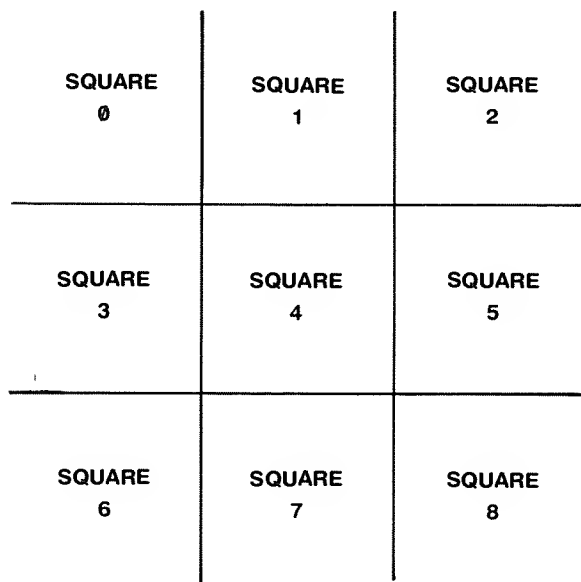
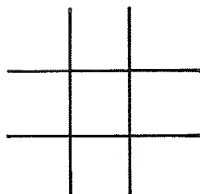


Figure 14-5. Tic-Tac-Toe Squares

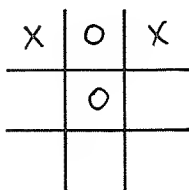
Sample configurations are shown in Figure 14-6. The beginning of a game has all blanks and is therefore represented by 0-0-0-0-0-0-0-0-0. A typical "middle game" shown in the figure has a combination of Xs, Os, and blanks and is represented by 1-2-1-0-2-0-0-0-0. A typical end game shown in the figure is represented by 2-1-0-2-2-0-1-1-1.



BEGINNING OF GAME

SQUARE

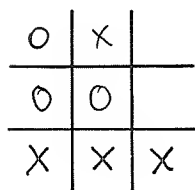
0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0



TYPICAL MIDDLE GAME

SQUARE

0	1	2	3	4	5	6	7	8
1	2	1	0	2	0	0	0	0



TYPICAL END GAME

SQUARE

0	1	2	3	4	5	6	7	8
2	1	0	2	2	0	1	1	1

Figure 14-6. Tic-Tac-Toe Configurations

Figuring Permutations

How many different combinations, or more precisely, **permutations** are there of Xs, Os, and spaces? We can find out by listing all permutations of the numbers. We'll start with 000000000, and end with 222222222. We've eliminated the dashes for compactness.

Wait, a minute, this looks suspiciously like a range of numbers, not binary, since there are **three** symbols, but **base three**. This is no more mysterious than binary! There are three symbols, 0, 1, and 2, and we count much the same way — 0,1,2,10,11,12,20,21,22, . . .

Base 3 Numbers

Base 3 numbers (ternary) use the same positional notation as binary, decimal, and hexadecimal. The rightmost digit is 3^0 , next is 3^1 , and so forth.

$$\begin{array}{rcl}
 212 & & \\
 \begin{array}{l} \text{---} \\ \text{---} \\ \text{---} \end{array} & \begin{array}{l} 2 \times 3^0 = 2 \\ 1 \times 3^1 = 3 \\ 2 \times 3^2 = 18 \end{array} & \\
 & & \underline{23}_{10}
 \end{array}$$

The same conversion techniques used in binary or hexadecimal can be used to convert between decimal and base three - double dabble:

$$\begin{array}{rcl}
 2021 & 2 \times 3 = 6 + 0 = 6 & \\
 \begin{array}{l} \text{---} \\ \text{---} \end{array} & \begin{array}{l} 6 \times 3 = 18 + 2 = 20 \\ 20 \times 3 = 60 + 0 = 60 \end{array} & \\
 & & \underline{61}_{10}
 \end{array}$$

and "divide by 3, save remainders":

$$\begin{array}{rcl}
 & 0 \text{ R2} & \\
 3 \overline{) 2} & \text{R0} & \\
 3 \overline{) 6} & \text{R2} & \\
 3 \overline{) 20} & \text{R1} & \\
 3 \overline{) 61} & & \\
 \hline
 & & 2021_3
 \end{array}$$

Just as a nine-digit binary number can hold 2 to the ninth permutations, a nine-digit base three number can hold 3 to the ninth permutations, or (let me use my pocket calculator here) 19,683 permutations.

One of those permutations will represent any arrangement of Xs, Os, and spaces we care to define. Many of the permutations are not possible in a game, such as 111111111, 222222222, 222222000, and many others!

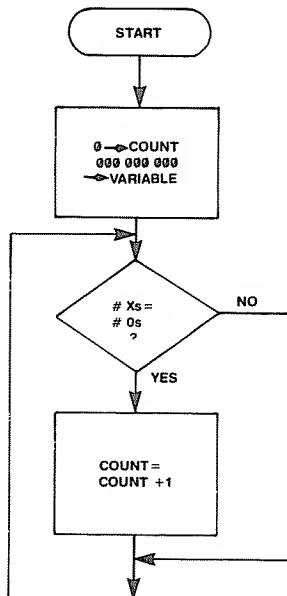
As a matter of fact, we can write a fairly simple assembly-language program to figure out the number of permutations we'll have to work with. 19,683

permutations are really too many to fit into memory with everything else. We'd like to see if we can reduce that number down to something manageable!

Reducing the Number of Permutations

We'll start by making some assumptions about the game to simplify things. First of all, the computer will always play first with an X. Secondly, we'll look for those permutations in which the computer is to play next, not in which the human is to play. We'll never have to deal with the latter case. This means that the number of Xs and Os have to be equal.

A flowchart for such a program is shown in Figure 14-7. It cycles a variable from 000000000 through 222222222 in base three. For each number, it tests the number of Xs (ones) and Os (twos). If they are equal, it adds 1 to a count. At the end, the count has the number of possible permutations of "computer to play next" (this includes 000000000 where the computer has not yet played, but $\#Xs = \#Os = 0$).



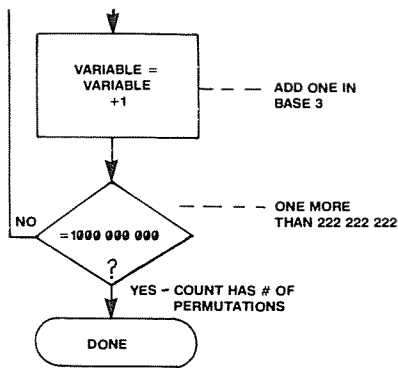
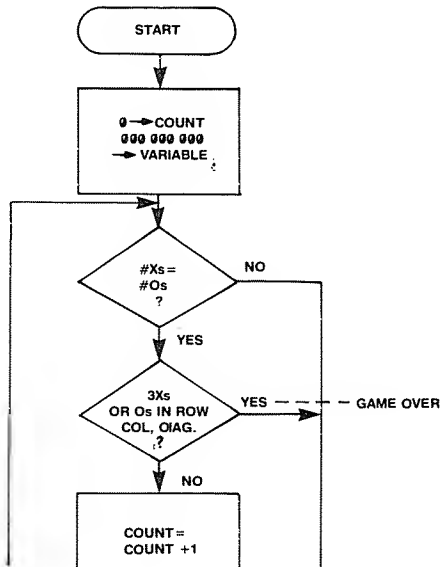


Figure 14-7. Flowchart Try # 1

When we run such a program, we find that there are about 3200 such permutations, a large reduction from all possible permutations of 19,683.

Still, we would like to reduce the permutations further. Well, we can delete all permutations where the game has already been won! We'd never continue from such a point in an actual game. This would be the case in which there are three Xs or Os in a row, column, or diagonal. The flowchart for such a program is shown in Figure 14-8.



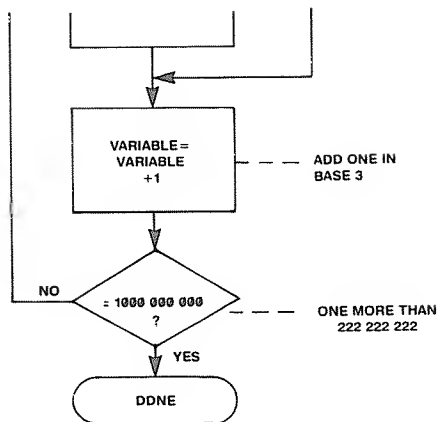


Figure 14-8. Flowchart Try # 2

When we implement the program, we find that there are now about 2460 valid permutations, a further reduction.

Another reduction we can make is to eliminate "9th move" permutations. In this case, 8 squares have been filled and the computer has only one choice. When this is added to the analysis program, we have cut down the allowable permutations by another 100 or so.

We can reduce further by eliminating permutations where there are two Xs in a row, column, or diagonal. This means the computer will win on the next move and knows enough to finish the game. This reduces the number of permutations we have to deal with still further.

Are there any other reductions possible? Yes, a major one. We know that there are many configurations of tic-tac-toe moves that are identical except for **rotation**. Figure 14-9 shows some of these. Also, there are identical games when "mirror images" are considered. We can cut down on the permutations we have to deal with drastically by considering rotations and mirror images.

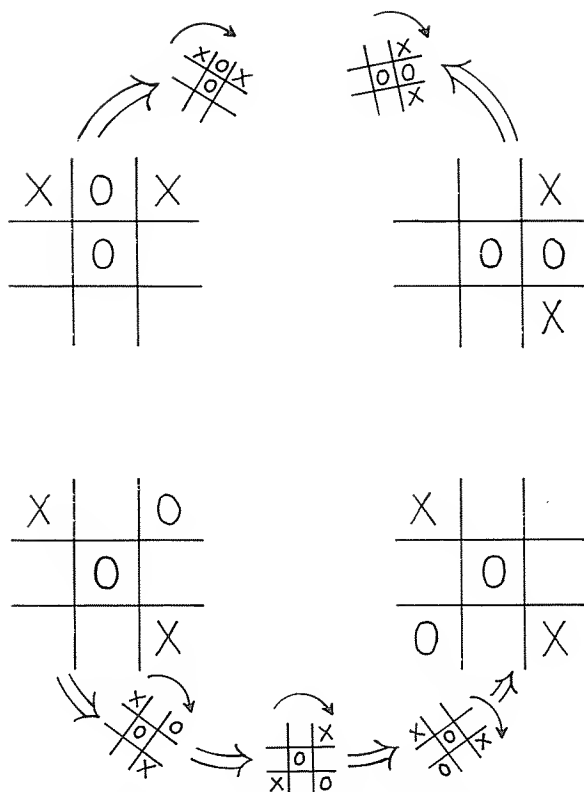
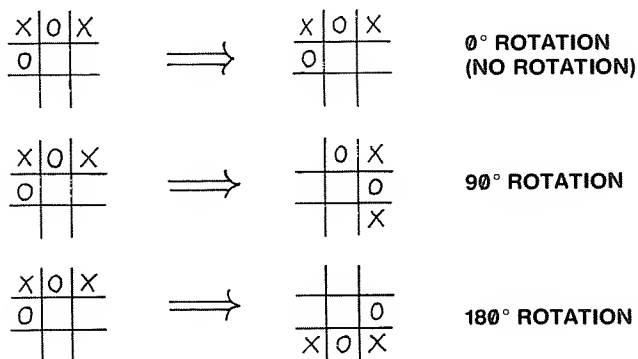
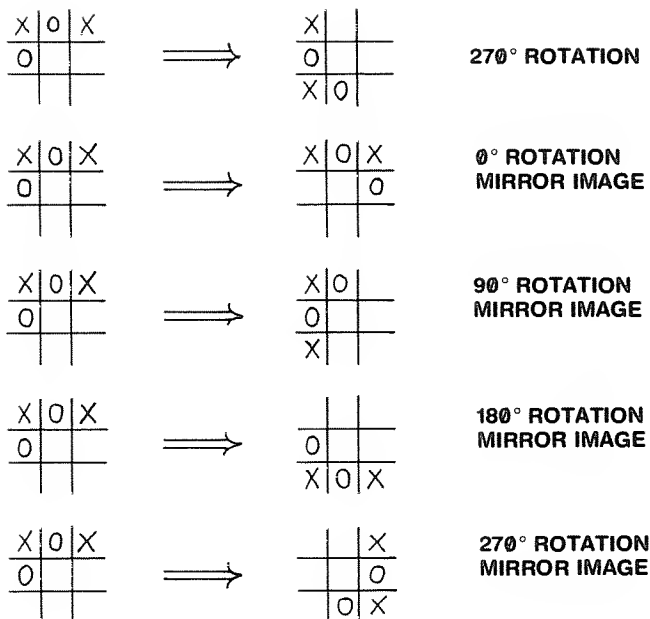


Figure 14-9. Rotation Identities

There are eight rotations and mirror images to consider, and they are shown in Figure 14-10.





**Figure 14-10. Rotations
and Mirror Images**

Our analysis program now eliminates:

1. All permutations in which the number of Xs and Os are not equal.
2. All permutations for the last move.
3. All permutations in which the game has already been completed.
4. All permutations in which the computer will win on the next move.
5. All permutations which are rotations or mirror images of other permutations already recorded.

The flowchart for figuring out the number of permutations we have to deal with is shown in Figure 14-11.

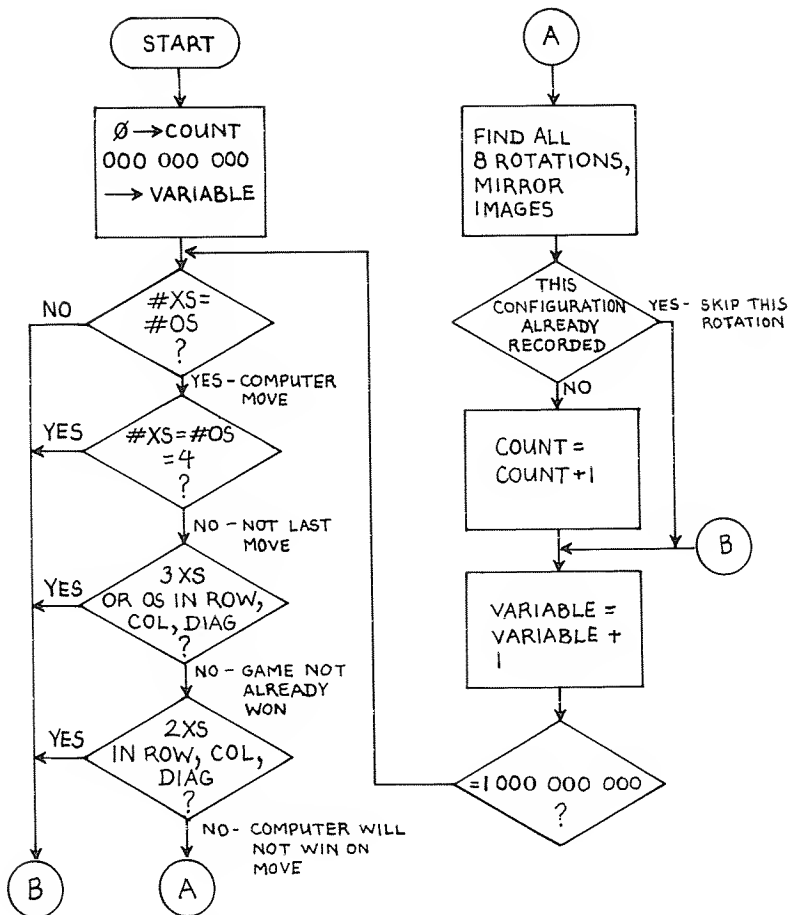


Figure 14-11. Final Flowchart

What do we really have after totaling the number of permutations in this fashion? We have a complete record of all configurations that we would encounter in any tic-tac-toe game in which the computer plays first and knows enough to complete a row, column, or diagonal; to recognize that the game has been lost; and to recognize that it will win on the next move. More importantly, the number of permutations we have to deal with have been reduced from 19,683 to 120! Now we can fit the permutations in memory and possibly implement the program!

Alternatives To Learning

Given a reasonable number of permutations to work with, how do we implement artificial intelligence in the program? How do we make it learn?

One approach would be to implement some sort of giant **binary tree** structure in the program as shown in Figure 14-12. We'd start off with the "empty" configuration and construct all possible courses a tic-tac-toe game could take. Then, if the computer lost, we would delete the last move so that the computer could never make that play again. When all possible lower branches were deleted, we'd delete the upper "limb." Given enough games, we'd have a computer that "learned" by never making the same mistake twice.

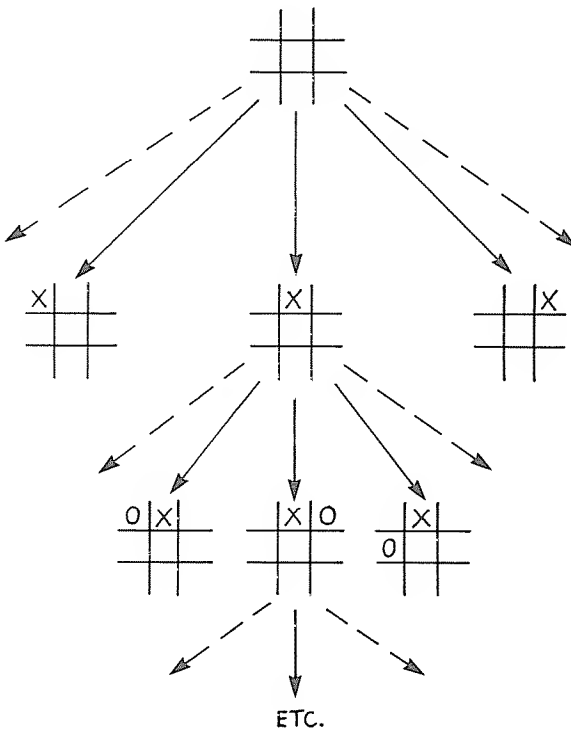


Figure 14-12. Tic-Tac-Toe
Binary Tree

Learning By Losing

If we implemented the 'learning by losing' algorithm, we'd have a very mechanical learning machine, and we'd have to play hundreds of games to make much progress.

What we really want is not so much an infallible 'idiot savant' as something that will emulate human learning — learning by trial and error with reinforcement of successful actions and rejection of unsuccessful approaches.

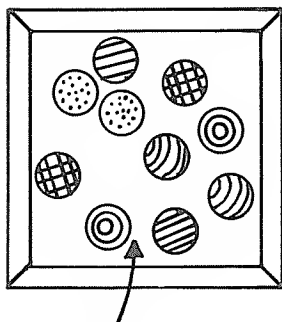
Another way to create a learning process was described in the pages of Scientific American some years ago, and attributed to work done by Michie at the University of Edinburgh. Suppose that we emulate the tree structure described above by a series of boxes. Each box is marked with the permutation that we've allowed by our analysis program.

Inside each box, we put different colored balls, each color representing one of the paths that could be taken. There could be nine colors for a "first move" configuration, seven for a "third move," five for a "fifth move," and so forth.

We'll put four balls of each color into first move boxes, three of each color into third move boxes, two of each color into fifth move boxes, and one of each color into seventh move boxes. By the ninth move the game will be over, and we have no ninth move boxes. This situation is shown in Figure 14-13.

X	O	X
3	O	5
6	7	8

**"FIFTH MOVE
CONFIGURATION"**



TWO BALLS OF
EACH COLOR FOR
FIFTH MOVE. ONE
BALL SELECTED
WHILE BLINDFOLDED.

**IF THIS BALL SELECTED,
COMPUTER:**






-  PUTS X IN SQUARE 3
-  PUTS X IN SQUARE 5
-  PUTS X IN SQUARE 6
-  PUTS X IN SQUARE 7
-  PUTS X IN SQUARE 8

Figure 14-13. Boxes and Balls

Now we'll play a game. For each move, we'll shake the box and withdraw one ball. We'll note the color and put it back in the box. Its color will determine in which square the "computer" will play. The human now makes a move. After the human's move, we go through the process again, choosing one ball at random from the next box representing the current configuration.

Eventually the "computer" wins, loses, or draws. If the computer wins, we'll go back and add three balls for each color chosen to each box. For example, if the first move box was a red, signifying square 2, we'll add three red balls. We'll also add three balls of the appropriate color to the third move, fifth move, etc., boxes.

If the "computer" loses, we'll take away one ball of the proper color from each of the boxes involved. If the "computer" draws, we'll add only one ball of the appropriate color.

If a particular color (which is really a number of a square) consistently wins, we'll start accumulating a higher and higher proportion of balls of the color (square) that produces winning games. Similarly, if a color consistently loses, we'll have fewer and fewer balls of that color.

Since one ball is chosen at random for each move, we'll have a better chance to withdraw winning colors (squares) as more games are played. Winning games "reward" the square choice, losing games "punish" the square choice, and draws "reward" the choice slightly. This emulates the reward and punishment of human learning for a mechanical process and is much more interesting and faster than just deleting losing paths. This is the method we've chosen to adopt in our implementation of tic-tac-toe on the TRS-80 in this program.

Algorithms

The **algorithms** we'll be using for the tic-tac-toe program are described in the following section. These algorithms emulate the tic-tac-toe learning procedure described above — the "reward" and "punishment" method. They are divided into two parts:

1. Generation of a table of permutations representing legitimate tic-tac-toe games, and
2. Algorithms for playing the game itself.

Generation of a Permutation Table

First of all we must generate a table of all the possible permutations the computer will encounter in playing the game of tic-tac-toe. As we described above, these represent configurations where the computer is to play: the so-called "first move," "third move," "fifth move," and "seventh move" conditions.

In generating the permutations, we'll discard those where the game has already been won or where there are two Xs (computer) in a row, column, or diagonal. We'll also look for the "rotations" and "mirror images" and pick only one out of eight possible, discarding the rest. At the end of the

table generation we'll have a table of hundreds of entries that represent any configuration that the computer will encounter when it is playing the game.

The algorithm for table generation is as follows:

1. Start with a value of the base three number 000 000 000. Increment this value by one each time through the steps up to a maximum of 222 222 222. This value is known as the "current value."
2. Start with the address of a table in memory known as the "permutation table" or "PTABLE". For each valid permutation, we'll enter the current value and some other data. We'll build up this table so at the end of table generation, we'll have a table in memory that holds all configurations.
3. Take the current value. Test for the end of 1 000 000 000 (one more than 222 222 222). If not equal, continue; otherwise the table is done.
4. Count number of Xs and Os in current value. If they are not equal, this is not first, third, fifth, seventh, or last move (computer's turn); if not equal, go on to the next current value.
5. Test for number of Xs = number of Os = 4. If this is true, this is the last move. The computer knows what to do here, so go on to the next current value.
6. Test for all Xs or Os in a row, column, or diagonal. (All ones or twos). If there are all ones or twos, discard this permutation as the game has already been won. Go on to the next current value.
7. Test for two Xs (ones) in a row, column, or diagonal. If this is true, the computer will know enough to finish the game as it is the computer's move; go on to the next current value.
8. Rotate (and find the mirror image) of the current value seven different ways. **Take the lowest base three value** and compare it to the current value. If they are equal, save the current value in the PTABLE, otherwise go on to the next current value.
9. At this point we have a "valid" current value, one that will be saved in the PTABLE. A sample current

value that reached this point is shown in Figure 14-14. Follow the next steps to "process" that value.

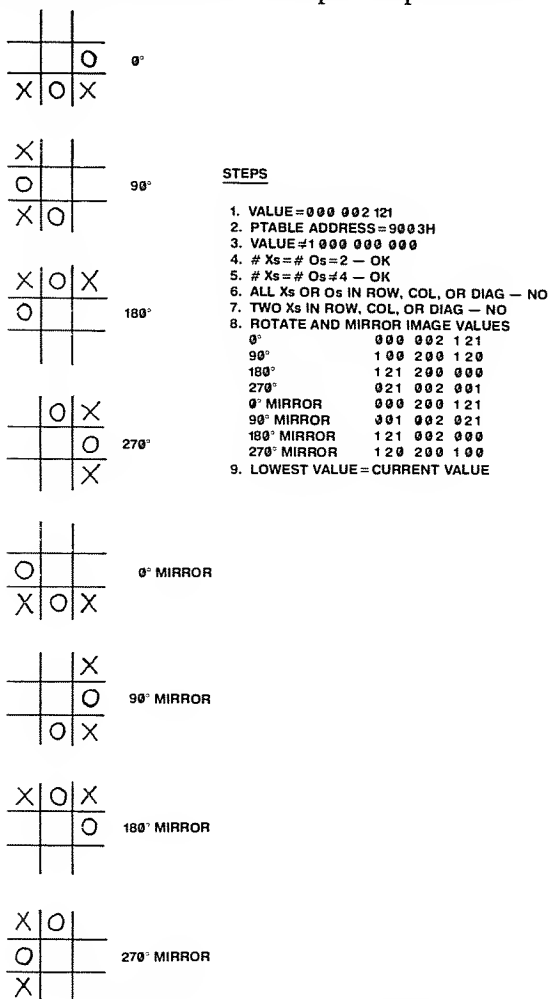


Figure 14-14. PTABLE Initial Processing

10. Enter the current value in the next position of the PTABLE.
11. Count the number of spaces in the current value (number of zeroes). Put this number in the next position of the PTABLE. This will range from 9 spaces for a first move to 3 for a seventh move.

12. Allocate a number of bytes equal to the number of spaces from 11.
13. Into each of the bytes, put a value of 4,3,2, or 1, for 9,7,5, or 3 spaces, respectively. In other words, if the number of spaces is 5, put a value of 2 into the two bytes allocated for the spaces. The entry for this permutation is now completed. The sample is shown in Figure 14-15.

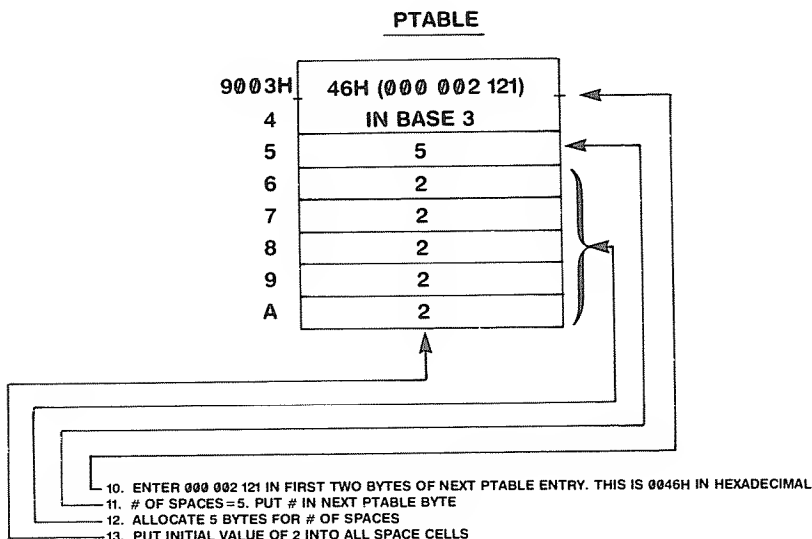


Figure 14-15. PTABLE Final Processing

What we've constructed in the PTABLE above is a computer analogy to the "boxes" we discussed earlier. Each box (entry of PTABLE) has a configuration associated with it represented by the current value. Each box (entry) has 9 to 3 "cells" (bytes), each representing a space or move the computer can make, reading left to right, top to bottom, the same way you would scan a page.

Into each space cell, we've put a value which is the same as putting in a certain number of colored balls. Considering all space cells together, we have a number of

balls, each representing a certain move. We'll add to or subtract from this count by changing the number in the space cells as we total up games. We'll withdraw one ball by choosing a random space cell of the 9 to 3 space cells available. We'll discuss this in more detail in the next section.

Algorithms for Playing the Game

Having established the "PTABLE", which represents any possible configuration that the computer will have to deal with, we'll now look at how the game is played by using the table to emulate the "boxes and balls" approach.

The algorithm goes like this:

1. Start with a blank array drawn on the screen with the usual grid.
2. Take the current array configuration and "analyze" it — count the number of Xs(computer), Os(human), and spaces. If entering this step from step one, the array will be all blanks.
3. See if the computer can complete a row, column, or diagonal of Xs. If so, perform the completion and display; the computer wins. Record the win and go to step 9. If not, continue.
4. See if this is the last move. If it is, the computer makes the only move available and outputs it to the screen. Then the computer analyzes the new configuration and sees whether it wins, loses, or draws and goes to step 9.
5. At this point, the PTABLE has not been referenced, but the computer has looked for obvious moves. Now the PTABLE will be referenced. The computer now performs all seven rotations to find which of the eight configurations should be found in the PTABLE. The PTABLE is then searched to find the proper one. This search must be successful, as the PTABLE holds all permutations!
6. Now the computer looks for two Os (human) in a row, column, or diagonal. If this condition is found, the

- computer "blocks" by putting an X in the proper space, displays the screen, and goes to step 8; otherwise, it continues.
7. At this point, no "block" was possible. The computer now "draws a ball" by randomly choosing **one** of the space cells from the PTABLE entry. It records the space cell number, puts an X in the array, displays the new array on the screen, and continues.
 8. Now the computer inputs the human choice. It checks for the proper choice, displays the new array, and analyzes the new configuration. If the human wins, it records the win and goes to step 9. If there is no win (there cannot be a draw), the computer goes back to step 2.
 9. This is the "end of game" step. The computer has a record of all configurations and which space cells were chosen for the computer's move. If the game was a win, it adds 3 "balls" to each of the space cells by increasing the count by three. This may require three adds for a five move game, four adds for a seven move game, or five adds for a nine move game. If the game was a loss, one "ball" is subtracted from the each of the space cells. If the game was a draw, one "ball" is added to each of the space cells. The computer now records the win/lose/draw for the record, and goes back to step 1 for a new game.

There is one slight addition to the above algorithm. If in step 7, all space cells were found to contain zeroes, the computer **concedes** and zeroes the **previous** space cell that resulted in the current permutation. In this case, the current configuration is considered so hopeless that it is discarded. This condition will rarely, if ever, happen.

Implementation

Modules

Tic-tac-toe is implemented as a series of five levels of modules as shown in Figure 14-16. The top levels of modules are the main **drivers** of the program, while the bottom level has the most rudimentary subroutines of the program.

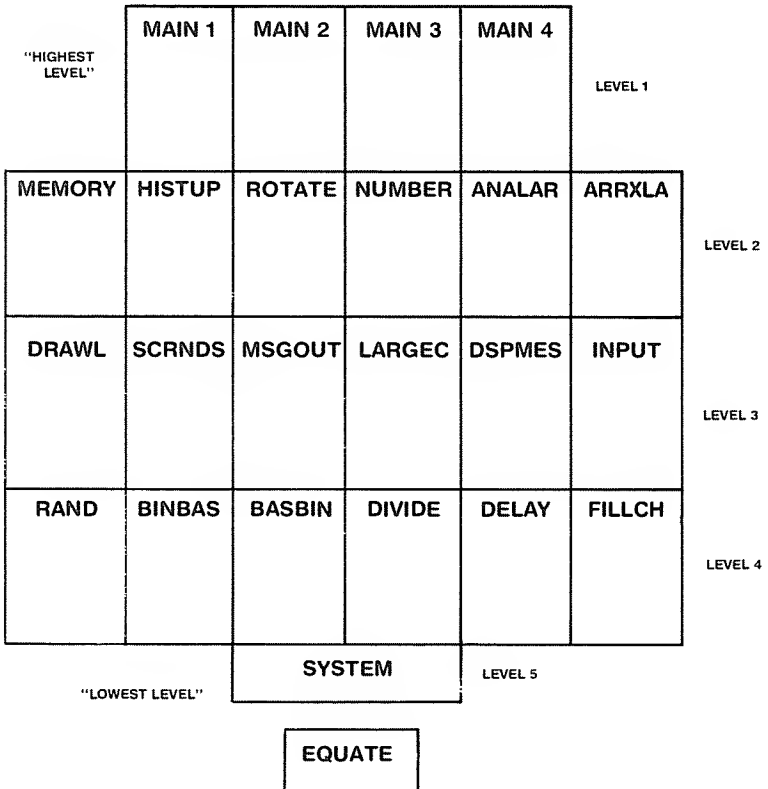


Figure 14-16. Tic-Tac-Toe Modules

As in the previous MORG program, each module is dedicated to a function applicable to the tic-tac-toe program or to a general application usable in programs such as division.

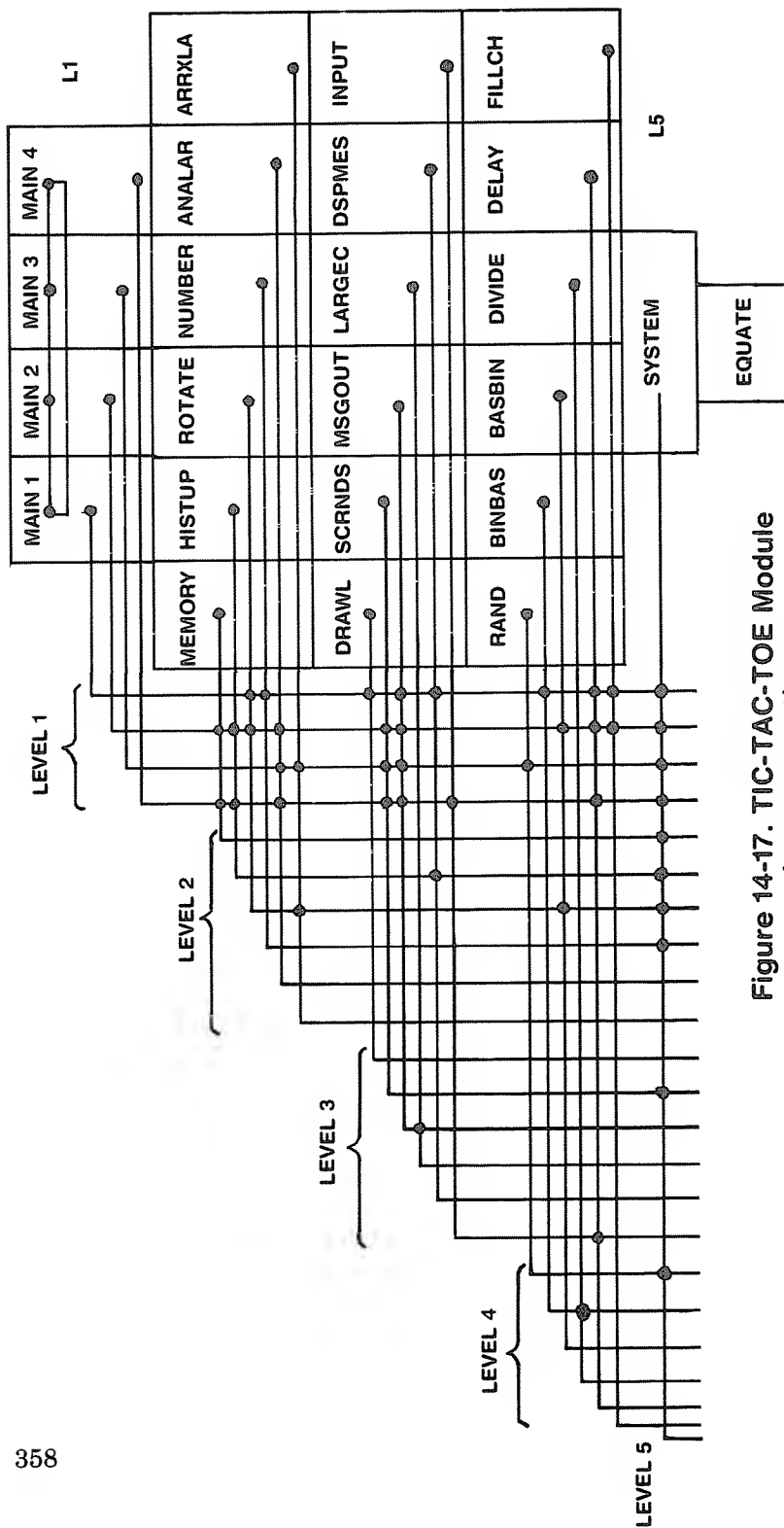


Figure 14-17. TIC-TAC-TOE Module Interconnections

Figure 14-17 shows the module interconnections. Each module generally calls a lower-level module by a CALL, with a set of parameters in the CPU registers. Communication is also done by the tables and variables in the SYSTEM module, which are commonly accessed by many different modules.

Hints and Kinks 14-3

Notes on Figure 14-17

Here, as in Figure 13-7, it's interesting to look at the interconnections to see how the modules relate to each other. Most of the action takes place in the level one modules, which call many of the other modules from all levels. One reason for this is that this program has heavy processing taking place in 'main-line' code of MAIN1 through MAIN4.

The flow of the program is sequential in these modules and couldn't easily be broken down functionally into lower-level subroutines. There's very little downward communication from level two to other modules.

To find the modules called by any specific module, follow the horizontal line from the module to the extreme left. When the line turns downward, read the lower-level module called by referencing the connection dots. The arrangement of the modules is duplicated in the program listing. Higher-level modules appear first, followed by lower-level modules.

Tables, Buffers, and Variables

Refer to the tic-tac-toe listings, Figure 14-18. There is one listing for each module. These were assembled on the Disk Editor/Assembler, but could have just as easily been done on EDTASM as one large program, providing memory requirements were not a factor in your system.

```

001000 TITLE EQUATE
001010 ENTRY SCREEN_SCARRY,LINE13,LASTL
001020 ;*****
001030 ; * SYSTEM EQUATES
001040 ;*****
001050 ;
001060 ;
001070 ;
001080 ;
001090 ;
001100 TITLE MAIN1
001110 ENTRY ARTIP
001120 EXT SCREEN_SCARRY,LINE13
001130 EXT NUMBER,ANALR,ROTATE,ARRXLA,RAND,INPUT
001140 EXT FILCH,DSPMES,DRAW,MSGOUT,DELAY,BINBAS
001150 EXT SCRDNS,HISTUP,MEMORY,BASBTN
001160 EXT GRIDTB,MOVSTB,HISTR,ARAX1,ARRAY2,ROITAB
001170 EXT ANALFTB,HINDT,HINDR
001180 EXT MOVENO,FRSTF,MXTHS,PTABLE,PRNDW,NOPTE
001190 EXT NOX,NOO,NOSP,ROTFTR
001200 EXT MSG1,MSG2,MSG3,MSG4,MSG5,MSG6,MSG7,MSG8,MSG9,MSG9
001210 EXT LASTL,MSG10,MSG11,STACK
001220 ;*****
001230 ; * TIC-TAC-TOE LEARNING PROGRAM
001240 ;*****
001250 ;
001260 ;*****START OR RESTART ENTRY POINT*****
001270 ;
001280 ;
001290 ;
001300 ARTIP: LD SP,STACK
001310 LD A,80H ;SET STACK
001320 LD DE,SCREEN ;GRAPHICS ALL OFF
001330 LD B,832 ;START OF SCREEN
001340 CALL FILCH ;13 LINES
001350 LD H,MSG1 ;FILL DISPLAY AREA
001360 LD BC,LINE13 ;HISTORY MESSAGE
001370 CALL DSPMES ;START OF HISTORY AREA
001380 ;
001390 ;
001400 ;
001410 ;
001420 ;
001430 ;
001440 ;
001450 ;
001460 ;
001470 ;
001480 ;
001490 ;
001500 ;
001510 ;
001520 ;
001530 ;
001540 ;
001550 ;
001560 ;
001570 ;
001580 ;
001590 ;
001600 ;
001610 ;
001620 ;
001630 ;
001640 ;
001650 ;
001660 ;
001670 ;
001680 ;
001690 ;
001700 ;
001710 ;
001720 ;
001730 ;
001740 ;
001750 ;
001760 ;
001770 ;
001780 ;
001790 ;
001800 ;
001810 ;
001820 ;
001830 ;
001840 ;
001850 ;
001860 ;
001870 ;
001880 ;
001890 ;
001900 ;
001910 ;
001920 ;
001930 ;
001940 ;
001950 ;
001960 ;
001970 ;
001980 ;
001990 ;
002000 ;
002010 ;
002020 ;
002030 ;
002040 ;
002050 ;
002060 ;
002070 ;
002080 ;
002090 ;
002100 ;
002110 ;
002120 ;
002130 ;
002140 ;
002150 ;
002160 ;
002170 ;
002180 ;
002190 ;
002200 ;
002210 ;
002220 ;
002230 ;
002240 ;
002250 ;
002260 ;
002270 ;
002280 ;
002290 ;
002300 ;
002310 ;
002320 ;
002330 ;
002340 ;
002350 ;
002360 ;
002370 ;
002380 ;
002390 ;
002400 ;
002410 ;
002420 ;
002430 ;
002440 ;
002450 ;
002460 ;
002470 ;
002480 ;
002490 ;
002500 ;
002510 ;
002520 ;
002530 ;
002540 ;
002550 ;
002560 ;
002570 ;
002580 ;
002590 ;
002600 ;
002610 ;
002620 ;
002630 ;
002640 ;
002650 ;
002660 ;
002670 ;
002680 ;
002690 ;
002700 ;
002710 ;
002720 ;
002730 ;
002740 ;
002750 ;
002760 ;
002770 ;
002780 ;
002790 ;
002800 ;
002810 ;
002820 ;
002830 ;
002840 ;
002850 ;
002860 ;
002870 ;
002880 ;
002890 ;
002900 ;
002910 ;
002920 ;
002930 ;
002940 ;
002950 ;
002960 ;
002970 ;
002980 ;
002990 ;
003000 ;
003010 ;
003020 ;
003030 ;
003040 ;
003050 ;
003060 ;
003070 ;
003080 ;
003090 ;
003100 ;
003110 ;
003120 ;
003130 ;
003140 ;
003150 ;
003160 ;
003170 ;
003180 ;
003190 ;
003200 ;
003210 ;
003220 ;
003230 ;
003240 ;
003250 ;
003260 ;
003270 ;
003280 ;
003290 ;
003300 ;
003310 ;
003320 ;
003330 ;
003340 ;
003350 ;
003360 ;
003370 ;
003380 ;
003390 ;
003400 ;
003410 ;
003420 ;
003430 ;
003440 ;
003450 ;
003460 ;
003470 ;
003480 ;
003490 ;
003500 ;
003510 ;
003520 ;
003530 ;
003540 ;
003550 ;
003560 ;
003570 ;
003580 ;
003590 ;
003600 ;
003610 ;
003620 ;
003630 ;
003640 ;
003650 ;
003660 ;
003670 ;
003680 ;
003690 ;
003700 ;
003710 ;
003720 ;
003730 ;
003740 ;
003750 ;
003760 ;
003770 ;
003780 ;
003790 ;
003800 ;
003810 ;
003820 ;
003830 ;
003840 ;
003850 ;
003860 ;
003870 ;
003880 ;
003890 ;
003900 ;
003910 ;
003920 ;
003930 ;
003940 ;
003950 ;
003960 ;
003970 ;
003980 ;
003990 ;
004000 ;
004010 ;
004020 ;
004030 ;
004040 ;
004050 ;
004060 ;
004070 ;
004080 ;
004090 ;
004100 ;
004110 ;
004120 ;
004130 ;
004140 ;
004150 ;
004160 ;
004170 ;
004180 ;
004190 ;
004200 ;
004210 ;
004220 ;
004230 ;
004240 ;
004250 ;
004260 ;
004270 ;
004280 ;
004290 ;
004300 ;
004310 ;
004320 ;
004330 ;
004340 ;
004350 ;
004360 ;
004370 ;
004380 ;
004390 ;
004400 ;
004410 ;
004420 ;
004430 ;
004440 ;
004450 ;
004460 ;
004470 ;
004480 ;
004490 ;
004500 ;
004510 ;
004520 ;
004530 ;
004540 ;
004550 ;
004560 ;
004570 ;
004580 ;
004590 ;
004600 ;
004610 ;
004620 ;
004630 ;
004640 ;
004650 ;
004660 ;
004670 ;
004680 ;
004690 ;
004700 ;
004710 ;
004720 ;
004730 ;
004740 ;
004750 ;
004760 ;
004770 ;
004780 ;
004790 ;
004800 ;
004810 ;
004820 ;
004830 ;
004840 ;
004850 ;
004860 ;
004870 ;
004880 ;
004890 ;
004900 ;
004910 ;
004920 ;
004930 ;
004940 ;
004950 ;
004960 ;
004970 ;
004980 ;
004990 ;
005000 ;
005010 ;
005020 ;
005030 ;
005040 ;
005050 ;
005060 ;
005070 ;
005080 ;
005090 ;
005100 ;
005110 ;
005120 ;
005130 ;
005140 ;
005150 ;
005160 ;
005170 ;
005180 ;
005190 ;
005200 ;
005210 ;
005220 ;
005230 ;
005240 ;
005250 ;
005260 ;
005270 ;
005280 ;
005290 ;
005300 ;
005310 ;
005320 ;
005330 ;
005340 ;
005350 ;
005360 ;
005370 ;
005380 ;
005390 ;
005400 ;
005410 ;
005420 ;
005430 ;
005440 ;
005450 ;
005460 ;
005470 ;
005480 ;
005490 ;
005500 ;
005510 ;
005520 ;
005530 ;
005540 ;
005550 ;
005560 ;
005570 ;
005580 ;
005590 ;
005600 ;
005610 ;
005620 ;
005630 ;
005640 ;
005650 ;
005660 ;
005670 ;
005680 ;
005690 ;
005700 ;
005710 ;
005720 ;
005730 ;
005740 ;
005750 ;
005760 ;
005770 ;
005780 ;
005790
```

```

CD 0000*      00450      CALL      ; DRAWL      ; OHAW LINE
0001*          00460      LD          BC,5      ; 5 BYTES PER LINE
0002*          00470      ADD         IX,BC      ; POINT TO NEXT LINE
0003*          00480      JR          ART005     ; GO FOR NEXT LINE
0004*          00490      ; ZERO MDVE TABLE
AF          00500      ART008: XDR A      ; 0
0005*          00510      LD          DE,MDVEVB ; MDVE TABLE ADDRESS
0006*          00520      LD          BC,8      ; 8 BYTES
0007*          00530      CALL        FILCH      ; ZERO MDVE TABLE
0008*          00540      LD          (MOVEND),A ; ZERO MDVE NUMBER
0009*          00550      ; TEST FDR FIRST TIME ACTIDNS
0010*          00560      LD          A,(FRSTF) ; GET FIRST TIME FLAG
0011*          00570      JP          A          ; TEST
0012*          00580      JP          NZ,MAIN2   ; GD IF NOT FIRST TIME
0013*          00590      PAGE
0014*          00600      ;*****
0015*          00610      ; FIRST TIME ACTIONS HERE
0016*          00620      ;*****
0017*          00630      ;*****
0018*          00640      INC          A          ; 1 TD A
0019*          00650      LD          (FRSTF),A ; SET FIRST TIME FLAG
0020*          00660      LD          DE,HISTRY-1 ; START DF HISTDRY LINE-1
0021*          00670      LD          (NATHIS),OE ; INITIALIZE HISTDRY POINTER
0022*          00680      INC          DE        ; START DF HISTDRY LINE
0023*          00690      LD          A, ' '      ; 8LANK
0024*          00700      LD          BC,128     ; 128 BLANKS
0025*          00710      CALL        FILCH      ; INITIALIZE HISTORY LINE
0026*          00720      LD          HL,MSG2   ; TITLE MESSAGE
0027*          00730      LD          IY,SCREEN ; SCREEN START
0028*          00740      CALL        MSGOUT    ; OUTPUT MESSAGE
0029*          00750      LD          HL,300D    ; 300D MS
0030*          00760      CALL        DELAY     ; DELAY 3 SECS
0031*          00770      LD          HL,MSG3   ; "WAIT DNE" MESSAGE
0032*          00780      LD          IY,SCREEN ; SCREEN START
0033*          00790      CALL        MSGOUT    ; OUTPUT MESSAGE
0034*          00800      ; FIRST TIME ACTIDN: COMPUTE ALL PDSSIBLE PERMUTATIONS
0035*          00810      ; TIC-TAC-TDE GAME (CDMPUTER TD MDVE NEXT, COMPUTER
0036*          00820      ; MOVES FIRST) OTHER THAN THOSE WITH WHICH COMPUTER
0037*          00830      ; KNDWS WHAT TD DO, KNDWS WHAT KNDWS ENOUGH AT START TD
0038*          00840      ; COMPLETE A RDW, CDLUMN, DR DIAGNAL.
0039*          00850      ;
0040*          00860      LD          HL,PTABLE  ; ADDRESS DF PERM TABLE
0041*          00870      PUSH        HL        ; SAVE
0042*          00880      LD          HL,0      ; ZERO
0043*          00890      LD          (NOPT),HL ; INITIALIZE COUNT DF PERM
0044*          00900      ;
0045*          00910      ;
0046*          00920      ;
0047*          00930      ;
0048*          00940      ;
0049*          00950      ;
0050*          00960      ;
0051*          00970      ;
0052*          00980      ;
0053*          00990      ;
0054*          01000      ;
0055*          01010      ;
0056*          01020      ;
0057*          01030      ;
0058*          01040      ;
0059*          01050      ;
0060*          01060      ;
0061*          01070      ;
0062*          01080      ;
0063*          01090      ;
0064*          01100      ;
0065*          01110      ;
0066*          01120      ;
0067*          01130      ;
0068*          01140      ;
0069*          01150      ;
0070*          01160      ;
0071*          01170      ;
0072*          01180      ;
0073*          01190      ;
0074*          01200      ;
0075*          01210      ;
0076*          01220      ;
0077*          01230      ;
0078*          01240      ;
0079*          01250      ;
0080*          01260      ;
0081*          01270      ;
0082*          01280      ;
0083*          01290      ;
0084*          01300      ;
0085*          01310      ;
0086*          01320      ;
0087*          01330      ;
0088*          01340      ;
0089*          01350      ;
0090*          01360      ;
0091*          01370      ;
0092*          01380      ;
0093*          01390      ;
0094*          01400      ;
0095*          01410      ;
0096*          01420      ;
0097*          01430      ;
0098*          01440      ;
0099*          01450      ;
0100*          01460      ;
0101*          01470      ;
0102*          01480      ;
0103*          01490      ;
0104*          01500      ;
0105*          01510      ;
0106*          01520      ;
0107*          01530      ;
0108*          01540      ;
0109*          01550      ;
0110*          01560      ;
0111*          01570      ;
0112*          01580      ;
0113*          01590      ;
0114*          01600      ;
0115*          01610      ;
0116*          01620      ;
0117*          01630      ;
0118*          01640      ;
0119*          01650      ;
0120*          01660      ;
0121*          01670      ;
0122*          01680      ;
0123*          01690      ;
0124*          01700      ;
0125*          01710      ;
0126*          01720      ;
0127*          01730      ;
0128*          01740      ;
0129*          01750      ;
0130*          01760      ;
0131*          01770      ;
0132*          01780      ;
0133*          01790      ;
0134*          01800      ;
0135*          01810      ;
0136*          01820      ;
0137*          01830      ;
0138*          01840      ;
0139*          01850      ;
0140*          01860      ;
0141*          01870      ;
0142*          01880      ;
0143*          01890      ;
0144*          01900      ;
0145*          01910      ;
0146*          01920      ;
0147*          01930      ;
0148*          01940      ;
0149*          01950      ;
0150*          01960      ;
0151*          01970      ;
0152*          01980      ;
0153*          01990      ;
0154*          02000      ;
0155*          02010      ;
0156*          02020      ;
0157*          02030      ;
0158*          02040      ;
0159*          02050      ;
0160*          02060      ;
0161*          02070      ;
0162*          02080      ;
0163*          02090      ;
0164*          02100      ;
0165*          02110      ;
0166*          02120      ;
0167*          02130      ;
0168*          02140      ;
0169*          02150      ;
0170*          02160      ;
0171*          02170      ;
0172*          02180      ;
0173*          02190      ;
0174*          02200      ;
0175*          02210      ;
0176*          02220      ;
0177*          02230      ;
0178*          02240      ;
0179*          02250      ;
0180*          02260      ;
0181*          02270      ;
0182*          02280      ;
0183*          02290      ;
0184*          02300      ;
0185*          02310      ;
0186*          02320      ;
0187*          02330      ;
0188*          02340      ;
0189*          02350      ;
0190*          02360      ;
0191*          02370      ;
0192*          02380      ;
0193*          02390      ;
0194*          02400      ;
0195*          02410      ;
0196*          02420      ;
0197*          02430      ;
0198*          02440      ;
0199*          02450      ;
0200*          02460      ;
0201*          02470      ;
0202*          02480      ;
0203*          02490      ;
0204*          02500      ;
0205*          02510      ;
0206*          02520      ;
0207*          02530      ;
0208*          02540      ;
0209*          02550      ;
0210*          02560      ;
0211*          02570      ;
0212*          02580      ;
0213*          02590      ;
0214*          02600      ;
0215*          02610      ;
0216*          02620      ;
0217*          02630      ;
0218*          02640      ;
0219*          02650      ;
0220*          02660      ;
0221*          02670      ;
0222*          02680      ;
0223*          02690      ;
0224*          02700      ;
0225*          02710      ;
0226*          02720      ;
0227*          02730      ;
0228*          02740      ;
0229*          02750      ;
0230*          02760      ;
0231*          02770      ;
0232*          02780      ;
0233*          02790      ;
0234*          02800      ;
0235*          02810      ;
0236*          02820      ;
0237*          02830      ;
0238*          02840      ;
0239*          02850      ;
0240*          02860      ;
0241*          02870      ;
0242*          02880      ;
0243*
```



```

0084' E5
0085' 01 4CE3
0086' B7
0087' ED 42
0088' CA U129'
0089' 09
0090' 08F'
0091' 0D 21 0000*
0092' CD 0000*
0093' 22 0000*
0094'

0099' 21 0000*
009C' C0 0000*
009D' 3A 0000*
009F' 57
00A2' 3A 0000*
00A3' BA
00A6' 28 03
00A7' E1
00A9' 18 79
00AA'

00AC'
00AE'

00B0'
00B1'
00B7' 06 08
00B9' 21 0000*
00BC' 7E 03
00BD' CA 00A9'
00BE' FE 0C
00C2' CA 00A9'
00C3' FE 02
00C7' CA 00A9'
00C9' 23
00CC' 10 ED
00CD'

0090U
0091U LD
0092U OR
0093U SBC
0094U JF
0095U AOD
0096U LO
0097U CALL
0098U LO
0099U ; COUNI NUMBER OF XS AND OS IN ARRAY. IF NOT EQUAL
0100U ; DISCARD THIS PERMUTATION.
0101U LO
0102U CALL
0103U LD
0104U LO
0105U LD
0106U CP
0107U JR
0108U ART014: POP
0109U HL
0109U JR
0110U
0111U ; # OF XS - # OF OS HERE. NOW TEST IF # XS = # YS = 4.
0112U ; COMPUTER ALWAYS KNOWS WHAT TO DO ON LAST MOVE!
0113U ART015: CP
0114U JR
0115U Z,ART014
0116U ; NOW TEST FOR ALL XS OR ALL YS IN A ROW, COLUMN, OR
0117U ; DIAGONAL. DISCARD THIS PERMUTATION IF GAME HAS ALREADY
0118U ; BEEN WON!
0119U ; ALSO TEST FOR TWO XS IN A ROW, COLUMN, OR DIAGONAL.
0120U ; COMPUTER KNOWS WHAT TO DO HERE!
0121U LD
0122U IX,ARRAY1
0123U ANALAR
0124U B,8
0125U L0
0126U HL,ANALTB
0127U L0
0128U CP
0129U Z,ART014
0130U CP
0131U JZ,ART014
0132U CP
0133U JZ,ART014
0134U INC
0135U HL
0136U ART020
0137U OJNZ
0138U ; TO CUT DOWN ON STORAGE REQUIREMENTS AND TO MAKE FOR
0139U ; FASTER LEARNING, ROTATE ARRAY 7 WAYS, MAKING TOTAL
0140U ; OF 8 POSITIONS.

```

```

00CF'      CD 0000*      01350      CALL ROTATE      ; ROTATE
00D2'      E1            01360      POP HL            ;GET ORIGINAL
00D3'      E5            01370      PUSH HL           ;SAVE CARRY
00D4'      B7            01380      OR A              ;CLEAR CARRY
00D5'      E0 52         01390      SBC HL,DE         ;COMPARE
00D7'      20 D0         01400      JR NZ,ART014       ; IF NOT LOWEST, DISCARD
                                ; WHEN I FINALLY HAVE VALUE IN HL OF ALLOWABLE PERMUTATION.
00D9'      E1            01420      POP HL            ;GET #
00DA'      DD E1         01430      POP IX            ;GET NEXT TABLE ADDRESS
00DC'      00 75 U0      01440      LD (IX),L         ;STORE LS BYTE
00DE'      00 74 01      01450      LD (IX+1),H       ;STORE MS BYTE
00E0'      DD 2A 0000*   01460      LD (IX+1),H       ;*****WARNING***** OP CODE FOR
00E1'      F0 23         01470      ;BE FDI SOME ASSEMBLER VERSIONS ERRONEOUSLY OUTPUT ODI
00E2'      F0 22 0000*   01480      ; IY,(NOTPE)
                                ; NOW FIND NUMBER OF SPACES IN PERMUTATION
00E3'      DD 2A 0000*   01490      L0 A,(NOPE)
00E4'      F0 23         01500      L0 C,A
00E5'      F0 22 0000*   01510      L0 (NOPE),IY
                                ; NOW FIND NUMBER OF SPACES IN PERMUTATION
00E6'      DD 2A 0000*   01520      L0 A,(NOPE)
00E7'      F0 23         01530      L0 C,A
00E8'      DD 2A 0000*   01540      L0 (IX+2),C
                                ; NOW FIND NUMBER OF SPACES IN PERMUTATION
00E9'      DD 71 02      01550      INC IX
00FA'      DD 23         01560      INC IX
00FB'      DD 23         01570      INC IX
00FC'      DD 23         01580      INC IX
00FD'      DD E5         01590      PUSH IX
00FE'      D1           01600      POP DE
00FF'      06 00        01610      L0 B,0
0100'      DD 09         01620      AD0 IX,BC
0101'      00 E5         01630      PUSH IX
0102'      79           01640      LD A,C
0103'      CB 3F        01650      SRL A
0104'      05           01660      PUSH OE
0105'      CD 0000*     01670      CALL FILLCH
0106'      3A 0000*     01680      L0 A,(NOTPE)
0107'      E6 01         01690      AN0 1
0108'      21 0000*     01700      LD HL,MSG10
0109'      28 03         01710      JR Z,ART04U
0110'      21 0000*     01720      L0 HL,MSG11
0111'      F0 21 0000*   01730      LD IY,LASTL
0112'      CD 0000*     01740      CALL MSGOUT
0113'      DD E1         01750      POP IX
0114'      DD E1         01760      L0 L,(IX-3)
0115'      DD 6E FD      01770      L0 H,(IX-2)
0116'      DD 66 FE      01780      ART05U: INC HL
0117'      C3 0084*     01790      JP ART012
0118'      E1           01800      ART060: POP HL
0119'

```

```

012A' 00 E1 01810 POP IX *****:TABLE POINTER*****
01820 *****
01830 *****MAIN2 MODULE FOLLOWS HERE*****
01840 *****
01850 MAIN2 EQU $ *****:JOURNEY*****
01860 ENO *****
01000 TITLE MAIN2 *****
01100 ENTRY MAINLP *****
01110 EXI SCREEN, SCARRY, LINE13 *****
01120 EXI HUBBER, ANALAR, ROTATE, ARRXLA, RANO, INPUT *****
01130 EXI FILCH, OSPNES, ORAWL, MSGOUT, OELAY, BINBAS *****
01140 EXI SCNNOS, HISIUP, MEWOR, BASBIN *****
01150 EXI GRIOTB, ROVETB, HISIRI, ARRA1, ARRAY2, ROTIAB *****
01160 EXI ANALTB, RINOT, RINOTR *****
01170 EXI MOVENO, FR5IF, MXIHIS, PTABLE, RINOW, NOPTE *****
01180 EXI NOX, MOO, NOSPF, ROTIFR *****
01190 EXI MSG1, MSG2, MSG3, MSG4, MSG5, MSG6, MSG7, MSG8, MSG9 *****
01200 EXI MAINEA *****
01210 *****
01220 *****ENTRY IF NOT FIRST TIME*****
01230 *****
01240 *****ART070: LO HL, MSG2 *****:TITLE MESSAGE*****
01250 *****LO I, SCREEN *****:SCREEN START*****
01260 *****MSGOUT *****:OUTPUT MESSAGE*****
01270 *****CALL HL, 3000 *****:3000 MS*****
01280 *****LO OELAY *****:OELAY 3 SECS*****
01290 *****CALL BC, 9 *****:FOR 9 ELEMENT*****
01300 *****LO XOR A *****:ZERO*****
01310 *****LO OE, ARRAY1 *****:ARRAY START*****
01320 *****CALL FILCH *****:ZERO ARRAY*****
01330 **********MAIN LOOP HERE*****
01340 *****
01350 *****MAINLP EQU $ *****:MOVE # ADDRESS*****
01360 *****LO HL, MOVENO *****:BUMP MOVE #*****
01370 *****INC (HL) *****:OBLIVIOUS COMPUTER MOVE - COMPLETING A*****
01380 *****: FIRST SEE IF *****
01390 *****: ROW, COLUMN, OR DIAGONAL *****
01400 *****ART100: LO IX, ARRAY1 *****:ARRAY ADDRESS*****
01410 *****CALL ANALAR *****:ANALYZE*****
01420 *****LO HL, ANALTB *****:START OF ROW, COL, OIAG*****
01430 *****LO B, 8 *****:8 VALUES*****
01440 *****LO A, (HL) *****:GET VALUE*****
01450 *****ART101: LO Z *****:TEST FOR 2 XS*****
01460 *****CP *****:GO IF 2 XS*****
01470 *****JR *****
01480 *****
0000' 21 0000# 0020' 00 21 0000#
0003' FO 21 0000# 0023'
0007' CO 0000# 0024'
000A' 21 0BB8 0025'
000C' CO 0000# 0026'
0010' 01 0009 0027'
0013' AF 0028'
0014' 11 0000# 002A'
0017' CO 0000# 002B'
001A' 21 0000# 002C'
001C' 34 002D'
001E' 00 21 0000# 002E'
0020' CO 0000# 002F'
0023' 21 0000# 0030'
0025' 06 06 0031'
0028' 7E 0032'
002A' FE 02 0033'
002B' 28 05 0034'
002C'
002D'
002E'
002F'
0030'
0031'
0032'
0033'
0034'
0035'
0036'
0037'
0038'
0039'
0040'
0041'
0042'
0043'
0044'
0045'
0046'
0047'
0048'
0049'
0050'
0051'
0052'
0053'
0054'
0055'
0056'
0057'
0058'
0059'
0060'
0061'
0062'
0063'
0064'
0065'
0066'
0067'
0068'
0069'
0070'
0071'
0072'
0073'
0074'
0075'
0076'
0077'
0078'
0079'
0080'
0081'
0082'
0083'
0084'
0085'
0086'
0087'
0088'
0089'
0090'
0091'
0092'
0093'
0094'
0095'
0096'
0097'
0098'
0099'
0100'
0101'
0102'
0103'
0104'
0105'
0106'
0107'
0108'
0109'
0110'
0111'
0112'
0113'
0114'
0115'
0116'
0117'
0118'
0119'
0120'
0121'
0122'
0123'
0124'
0125'
0126'
0127'
0128'
0129'
0130'
0131'
0132'
0133'
0134'
0135'
0136'
0137'
0138'
0139'
0140'
0141'
0142'
0143'
0144'
0145'
0146'
0147'
0148'
0149'
0150'
0151'
0152'
0153'
0154'
0155'
0156'
0157'
0158'
0159'
0160'
0161'
0162'
0163'
0164'
0165'
0166'
0167'
0168'
0169'
0170'
0171'
0172'
0173'
0174'
0175'
0176'
0177'
0178'
0179'
0180'
0181'
0182'
0183'
0184'
0185'
0186'
0187'
0188'
0189'
0190'
0191'
0192'
0193'
0194'
0195'
0196'
0197'
0198'
0199'
0200'
0201'
0202'
0203'
0204'
0205'
0206'
0207'
0208'
0209'
0210'
0211'
0212'
0213'
0214'
0215'
0216'
0217'
0218'
0219'
0220'
0221'
0222'
0223'
0224'
0225'
0226'
0227'
0228'
0229'
0230'
0231'
0232'
0233'
0234'
0235'
0236'
0237'
0238'
0239'
0240'
0241'
0242'
0243'
0244'
0245'
0246'
0247'
0248'
0249'
0250'
0251'
0252'
0253'
0254'
0255'
0256'
0257'
0258'
0259'
0260'
0261'
0262'
0263'
0264'
0265'
0266'
0267'
0268'
0269'
0270'
0271'
0272'
0273'
0274'
0275'
0276'
0277'
0278'
0279'
0280'
0281'
0282'
0283'
0284'
0285'
0286'
0287'
0288'
0289'
0290'
0291'
0292'
0293'
0294'
0295'
0296'
0297'
0298'
0299'
0300'
0301'
0302'
0303'
0304'
0305'
0306'
0307'
0308'
0309'
0310'
0311'
0312'
0313'
0314'
0315'
0316'
0317'
0318'
0319'
0320'
0321'
0322'
0323'
0324'
0325'
0326'
0327'
0328'
0329'
0330'
0331'
0332'
0333'
0334'
0335'
0336'
0337'
0338'
0339'
0340'
0341'
0342'
0343'
0344'
0345'
0346'
0347'
0348'
0349'
0350'
0351'
0352'
0353'
0354'
0355'
0356'
0357'
0358'
0359'
0360'
0361'
0362'
0363'
0364'
0365'
0366'
0367'
0368'
0369'
0370'
0371'
0372'
0373'
0374'
0375'
0376'
0377'
0378'
0379'
0380'
0381'
0382'
0383'
0384'
0385'
0386'
0387'
0388'
0389'
0390'
0391'
0392'
0393'
0394'
0395'
0396'
0397'
0398'
0399'
0400'
0401'
0402'
0403'
0404'
0405'
0406'
0407'
0408'
0409'
0410'
0411'
0412'
0413'
0414'
0415'
0416'
0417'
0418'
0419'
0420'
0421'
0422'
0423'
0424'
0425'
0426'
0427'
0428'
0429'
0430'
0431'
0432'
0433'
0434'
0435'
0436'
0437'
0438'
0439'
0440'
0441'
0442'
0443'
0444'
0445'
0446'
0447'
0448'
0449'
0450'
0451'
0452'
0453'
0454'
0455'
0456'
0457'
0458'
0459'
0460'
0461'
0462'
0463'
0464'
0465'
0466'
0467'
0468'
0469'
0470'
0471'
0472'
0473'
0474'
0475'
0476'
0477'
0478'
0479'
0480'
0481'
0482'
0483'
0484'
0485'
0486'
0487'
0488'
0489'
0490'
0491'
0492'
0493'
0494'
0495'
0496'
0497'
0498'
0499'
0500'
0501'
0502'
0503'
0504'
0505'
0506'
0507'
0508'
0509'
0510'
0511'
0512'
0513'
0514'
0515'
0516'
0517'
0518'
0519'
0520'
0521'
0522'
0523'
0524'
0525'
0526'
0527'
0528'
0529'
0530'
0531'
0532'
0533'
0534'
0535'
0536'
0537'
0538'
0539'
0540'
0541'
0542'
0543'
0544'
0545'
0546'
0547'
0548'
0549'
0550'
0551'
0552'
0553'
0554'
0555'
0556'
0557'
0558'
0559'
0560'
0561'
0562'
0563'
0564'
0565'
0566'
0567'
0568'
0569'
0570'
0571'
0572'
0573'
0574'
0575'
0576'
0577'
0578'
0579'
0580'
0581'
0582'
0583'
0584'
0585'
0586'
0587'
0588'
0589'
0590'
0591'
0592'
0593'
0594'
0595'
0596'
0597'
0598'
0599'
0600'
0601'
0602'
0603'
0604'
0605'
0606'
0607'
0608'
0609'
0610'
0611'
0612'
0613'
0614'
0615'
0616'
0617'
0618'
0619'
0620'
0621'
0622'
0623'
0624'
0625'
0626'
0627'
0628'
0629'
0630'
0631'
0632'
0633'
0634'
0635'
0636'
0637'
0638'
0639'
0640'
0641'
0642'
0643'
0644'
0645'
0646'
0647'
0648'
0649'
0650'
0651'
0652'
0653'
0654'
0655'
0656'
0657'
0658'
0659'
0660'
0661'
0662'
0663'
0664'
0665'
0666'
0667'
0668'
0669'
0670'
0671'
0672'
0673'
0674'
0675'
0676'
0677'
0678'
0679'
0680'
0681'
0682'
0683'
0684'
0685'
0686'
0687'
0688'
0689'
0690'
0691'
0692'
0693'
0694'
0695'
0696'
0697'
0698'
0699'
0700'
0701'
0702'
0703'
0704'
0705'
0706'
0707'
0708'
0709'
0710'
0711'
0712'
0713'
0714'
0715'
0716'
0717'
0718'
0719'
0720'
0721'
0722'
0723'
0724'
0725'
0726'
0727'
0728'
0729'
0730'
0731'
0732'
0733'
0734'
0735'
0736'
0737'
0738'
0739'
0740'
0741'
0742'
0743'
0744'
0745'
0746'
0747'
0748'
0749'
0750'
0751'
0752'
0753'
0754'
0755'
0756'
0757'
0758'
0759'
0760'
0761'
0762'
0763'
0764'
0765'
0766'
0767'
0768'
0769'
0770'
0771'
0772'
0773'
0774'
0775'
0776'
0777'
0778'
0779'
0780'
0781'
0782'
0783'
0784'
0785'
0786'
0787'
0788'
0789'
0790'
0791'
0792'
0793'
0794'
0795'
0796'
0797'
0798'
0799'
0800'
0801'
0802'
0803'
0804'
0805'
0806'
0807'
0808'
0809'
0810'
0811'
0812'
0813'
0814'
0815'
0816'
0817'
0818'
0819'
0820'
0821'
0822'
0823'
0824'
0825'
0826'
0827'
0828'
0829'
0830'
0831'
0832'
0833'
0834'
0835'
0836'
0837'
0838'
0839'
0840'
0841'
0842'
0843'
0844'
0845'
0846'
0847'
0848'
0849'
0850'
0851'
0852'
0853'
0854'
0855'
0856'
0857'
0858'
0859'
0860'
0861'
0862'
0863'
0864'
0865'
0866'
0867'
0868'
0869'
0870'
0871'
0872'
0873'
0874'
0875'
0876'
0877'
0878'
0879'
0880'
0881'
0882'
0883'
0884'
0885'
0886'
0887'
0888'
0889'
0890'
0891'
0892'
0893'
0894'
0895'
0896'
0897'
0898'
0899'
0900'
0901'
0902'
0903'
0904'
0905'
0906'
0907'
0908'
0909'
0910'
0911'
0912'
0913'
0914'
0915'
0916'
0917'
0918'
0919'
0920'
0921'
0922'
0923'
0924'
0925'
0926'
0927'
0928'
0929'
0930'
0931'
0932'
0933'
0934'
0935'
0936'
0937'
0938'
0939'
0940'
0941'
0942'
0943'
0944'
0945'
0946'
0947'
0948'
0949'
0950'
0951'
0952'
0953'
0954'
0955'
0956'
0957'
0958'
0959'
0960'
0961'
0962'
0963'
0964'
0965'
0966'
0967'
0968'
0969'
0970'
0971'
0972'
0973'
0974'
0975'
0976'
0977'
0978'
0979'
0980'
0981'
0982'
0983'
0984'
0985'
0986'
0987'
0988'
0989'
0990'
0991'
0992'
0993'
0994'
0995'
0996'
0997'
0998'
0999'

```

```

002F' 23          INC          HL          ;BUMP POINTER
0030' 10 F8        OJNZ         ART101     ;CONTINUE IF NOT 8
0032' 18 3F        JR          ART109     ;NO COMPLETION POSSIBLE
; COMPUTER WINS! COMPLETE THE ROW, COLUMN, OR DIAGONAL
; AND END GAME
0034' E5          ART102: PUSH HL
0035' 21 0000*    ART103: LD   HL,ARRAY1
0036' 7E          A,(HL)
0037' B7          OR          A
0039' 28 03        JR          Z,ART105
003A' 23          ART104: INC  HL
003D' 18 F9        JR          ART103
003F' E5          ART105: PUSH HL
0040' 3C          INC          A
0041' 77          LD          (HL),A
0042' DD 21 0000* LD          IX,ARRAY1
0046' CD 0000*    CALL        ANALAR
0049' E1          POP         HL
004A' DD E1        POP         IX
004C' DD 7E 00    LD          A,(IX)
004F' FE 03        CP          3
0051' 28 06        JR          Z,ART106
0053' AF          XOR          A
0054' 77          LD          (HL),A
0055' DD E5        PUSH        IX
0057' 18 E3        JR          ART104
0059' CD 0000*    CALL        SCRD5
005C' 3E 57        LD          A,'W'
005E' CD 0000*    CALL        H1STUP
0061' 3E 03        LD          A,3
0063' CD 0000*    CALL        MEMORY
0066' 21 0000*    LD          HL,MSG8
0068' FD 21 0000* LD          IX,SCREEN
006B' CD 0000*    CALL        MSGOUT
0070' C3 0000*    JP          MAINEA
; NOW TEST FOR LAST MOVE
0073' 3A 0000*    ART109: LD   A,(MOVENO)
0076' FE 05        CP          5
0078' 20 3A        JR          NZ,ART114
; LAST MOVE HERE
007A' 21 0000*    LD          HL,ARRAY1
007D' 7E          A,(HL)
007E' B7          OR          A
007F' 28 03        JR          Z,ART111
0081' 23          INC          HL
0082' 18 F9        JR          ART110
0083' 18 F9        JR          ART110
;BUMP POINTER
;CONTINUE

```



```

0042' 23 00560 INC HL ;BUMP POINTER
0043' 10 F8 00570 DJNZ ;CONTINUE IF NOT 8
0045' 18 52 00580 JR ART125 ;NO COMPLETION POSSIBLE
; TWO OS FOUND - PUT IN X FOR BLOCK
0047' E5 00590 ART102: PUSH HL ;SAVE ANALYZE VALUE
0048' 21 00000 00600 LD HL,ARRAY2 ;ARRAY ADDRESS
004B' 7E 00610 LD A,(HL) ;GET ELEMENT
004C' B7 00620 OR A ;TEST FOR 0
004D' 28 03 00630 JR Z,ART105 ;GO IF SPACE
004F' 23 00640 JR ART104 ;BUMP PTR
0050' 18 F9 00650 JR ART103 ;TRY NEXT
0051' 21 00660 ART105: PUSH HL ;SAVE ELEMENT ADDRESS
0052' E5 00670 INC A ;1 TO A
0053' 3C 00680 LD (HL),A ;STORE X
0054' 77 00690 LD IX,ARRAY2 ;ARRAY ADDRESS
0055' DD 21 00000 00700 CALL ANALAR ;ELEMENT ADDRESS
0059' CD 00000 00710 POP IX ;ELEMENT ADDRESS
005C' E1 00720 POP IX ;VALUE ADDRESS
005D' DD E1 00730 POP IX ;GET NEW TOTAL
005F' DD 7E 00 00740 LD A,(IX) ;COMPLETE?
0062' FE 09 00750 CP 9 ;GO IF YES
0064' 28 06 00760 JR Z,ART106 ;ZERO
0066' AF 00770 XOR A ;ZERO ELEMENT AGAIN
0067' 77 00780 LD (HL),A ;FOR LOOP
0068' DD E5 00790 PUSH IX ;MUST BE FOUND!
006A' 18 E3 00800 JR ART104 ;ZERO A
006C' AF 00810 ART106: XOR A ;ZERO FOR NEXT PROCESSING
006D' 77 00820 LD (HL),A ;SAVE ELEMENT ADDRESS
006E' E5 00830 PUSH HL ;ARRAY ADDRESS-1
006F' 01 FFFF0 00840 LD BC,ARRAY2-1 ;CLEAR C
0072' B7 00850 OR A ;INDEX OF ELEMENT+1
0073' ED 42 00860 SEC HL,BC ;ZERO A
; HL NOW CONTAINS INDEX OF "BLOCK" ELEMENT
0075' 11 FFFF 00870 LD DE,-1 ;COUNT
0078' 45 00880 LD B,L ;INDEX+1 TO B
0079' 21 00000 00890 LD HL,ARRAY2 ;ADDRESS OF ARRAY
007C' 7E 00900 LD A,(HL) ;GET ELEMENT
007D' 23 00910 ART108: LD HL ;BUMP TO NEXT
007E' B7 00920 OR A ;TEST FOR ZERO
007F' 20 01 00930 JR NZ,ART110 ;BYPASS IF NOT ZERO
0081' 13 00940 INC DE ;BUMP COUNT OF SPACES
0082' 10 F8 00950 ART110: DJNZ ART108 ;COUNT DOWN IF ZERO
; DE NOW HAS INDEX OF ELEMENT CELL
0084' E1 00960 POP HL ;GET ADDRESS OF ELEMENT
0085' 3E 01 00970 LD A,1 ;1 TO A
0087' 77 00980 LD (HL),A ;STORE X IN ARRAY2
0088' DD E1 00990 POP IX ;GET PERM TABLE ADDRESS

```

```

008A' 00 19 A0D IX,OE
008C' 0D 23 INC IX
008E' 00 23 INC IX
0090' 0D 23 INC IX
0092' 00 23 INC IX
0094' 00 E5 PUSH IX
0096' C3 0116' JR ART172
; FIND TOTAL COUNT IN ALL SPACES
0099' 00 E1 ART125: POP IX
009B' 00 E5 POP IX
009D' 00 46 02 ART130: LO B,(IX+2)
00A0' 21 0000' LO HL,0
00A3' 00 5E 03 ART135: LO E,(IX+3)
00A6' 16 00 LD 0,0
00A8' 19 01160 A00 HL,OE
00A9' D0 23 INC IX
00AB' 10 F6 DJNZ ART135
00AC' 7C LO A,H
00AE' B5 OR L
00AF' 00 E1 POP IX
00B1' 00 E5 PUSH IX
00B3' 20 25 JR NZ,ART140
; ALL SPACE CELLS OF THIS ENTRY ARE 0. REMOVE THIS
; ENTRY BY ZEROING PREVIOUS LINK AND CONCREOE.
00B5' 3A 0000* LO A,(MOVENO)
00B8' 3D OEC A
00B9' 07 RLCA
00BA' AF C,A
00BB' 06 00 LD B,0
00BD' D0 21 0000* LO IX,MOVETB
00C1' 0D 09 AD0 IX,BC
00C3' DD 66 00 LO H,(IX)
00C6' 00 6E 01 LO L,(IX+1)
00C9' AF XOR A
00CA' 77 01360 LO (HL),A
00CB' 77 01370 HL,MSG12
00CC' F0 21 0000* LO IY,SCREEN
00CE' C0 0000* CALL MSGOUT
00D1' 3E 43 LO A,C'
00D3' 01410 JP MAIN1
00D4' C3 0000* JP RAND
00D5' C5 01430 BC
00DE' 01 ART140: CALL RAND
00DF' EB EX OE,HL
00E0' B7 01450 OR A
00E1' E0 52 ART150: OR A
; RAND# IN HL, COUNT IN DE
; CLEAR CARRY
; RAND#-COUNT
;IX NOW PNTS TO SP CELL-3
;NOW SP CELL A00+1
;SAVE FOR NEXT PROCESSING
;NOW GO TO MOVE, OISPLAY
;GET PERM TABLE ADDR
;SAVE # OF SPACES
;CLEAR TOTAL
;GET COUNT
;NOW IN OE
;SUBTOTAL IN HL
;BUMP TO NEXT ENTRY
;GO IF NOT ENO
;GET COUNT MSB
;MERGE LSB
;GET POINTER
;SAVE
;GO IF NOT ZERO
;ALL SPACE CELLS OF THIS ENTRY ARE 0. REMOVE THIS
;ENTRY BY ZEROING PREVIOUS LINK AND CONCREOE.
;GET MOVE #
;FIND LAST MOVE INDEX
;INDEX#2
;NOW IN C
;NOW IN BC
;MOVE TABLE
;POINT TO LINK ADDRESS
;MS BYTE
;LS BYTE
;ZERO
;ZERO LINK
;I CONCREOE" MESSAGE
;MESSAGE AREA
;OISPLAY
;C FOR CONCREOE
;GO FOR END ACTION
;GET RANDOM #
;TRANSFER TO OE
;RAND# IN HL, COUNT IN DE
;CLEAR CARRY
;RAND#-COUNT

```



```

00E3' 30 FB
00E5' 19

01480
01490
01500
01510
01520
01530
01540
01550
01560
01570
01580
01590
01600
01610
01620
01630
01640
01650
01660
01670
01680
01690
01700
01710
01720
01730
01740
01750
01760
01770
01780
01790
01800
01810
01820
01830
01840
01850
01860
01870
01880
01890
01900
01910
01920
01930

30 FB
19

DD 4E 03
06 D0
B7
ED 42
D0 23
0DF0' 28 02
00F2' 30 F2

D1
DD 05
00F5' DD E5
00F7' E1
00F8' 01 0003
00FB' DD 09
DD E5
00FF' B7
E0 52
010D' E0 52
0102' 45
0103' DD 21 000D#
0107' DD 7E 00
010A' B7
010B' D0 23
010D' 20 F8
010F' 10 F6
0111' 3E 01
0113' DD 77 FF
0116' 3A 0000#
0119' 30
011A' 07
011B' 5F
011C' 16 00
011E' 21 0000#
0121' 19
0122' D1
0123' 1B
0124' 73
0125' 23
0126' 72

2A 0000#
DD 21 0000#

;CONTINUE TILL NEGATIVE
;GET # LE COUNT
; NDW HAVE # LESS THAN DR EQUAL TD TOTAL COUNT IN SPACE
; CELLS. USE THIS NUMBER TD FINO A RANDOM SPACE IN WHICH
; TD PUT AN X.
NC,ART150
HL,DE
; GET CDUNT
; NOW IN BC
; CLEAR CARRY
; #CDUNT
; BUMP PDINTER
; GD IF 0
; GD IF NDT NEGATIVE
; GET START
; TRANSFER CELL ADDRESS PNTR
; ADJUSTMENT
; PDINT TO CELL
; CELL ADDRESS+1
; CLEAR CARRY
; FIND # OF CELLS+4
; # DF CELL+1 TD B
; START OF WDRKING ARRAY
; GET ELEMENT
; TEST FDR 0
; BUMP POINTER
; GD IF NDT 0
; GO IF NDT ELEMENT
; X
; STDRX
; COMPUTER MDVE #
; MDVE #-1
; INDEX#2
; NDW IN E
; NDW IN OE
; MDVE TABLE ADDRESS
; PDINI TD MDVE #
; CELL ADDRESS+1
; ADDRESS DF SPACE
; STDRX LSB
; STDRX MSB
; NOW HAVE STORED AN X IN WDRKING ARRAY - CONVERT TO
; ACTUAL ARRAY.
; PDINTER TO INDICES LIST
; SOURCE ARRAY

```



```

0014' F5
0015' 8C
0016' CA UU1C'
0019' F2 001D'
001C' AF
001D' FE 64
001F' FA 0024'
0022' 3E 63
0024' 77
0025' DD 23
0027' DD 23
0029' F1
002A' 18 0C
002C' F1
002D' D0 E1
002F' E1
0030' F1
0031' C9

00330
00340 A00
00350 JP
00360 JP
00370 MEM008: XOR A
00380 MEM010: CP 100
00390 N, MEM020U
00390 A, 99
00400 (HL), A
00410 L0
00420 INC IX
00430 INC IX
00440 POP AF
00450 JR MEM005
00460 MEM090: POP AF
00470 POP IX
00480 POP HL
00490 POP AF
00500 RET
00510 END
00100 TITLE HISTSR
00110 ENTRY
;*****
; * NXTHIS, HISTRY, MSG1, LINE13, DSPMES
; *
; * HISTORY UPDATE
; *
; * UPDATES 128 PRINT POSITIONS OF HISTORY MESSAGE
; *
; * ENTRY: (A)='L', 'W', OR 'D' FOR LOSE, WIN, DRAW
; *
; * EXIT: BUFFER UPDATED
; *
; * ALL REGISTERS SAVED
; *
;*****
00200
00210
00220
00230
00240 LD HL, (NXTHIS)
00250 INC HL
00260 LD BC, HISTRY+128
00270 LD BC, HISTRY+128
00280 OR A
00290 SBC HL, BC
00300 JR NZ, HIS010
00310 DEC BC
00320 LD HL, (NXTHIS), BC
00330 LD HL, HISTRY+1
00340 LD DE, HISTRY
00350 LD BC, 127
00360 LDIR

0000' C5
0001' D5
0002' E5
0003' 2A 0000*
0006' 23
0007' 22 0000*
000A' 01 0080*
000D' B7
000E' ED 42
0010' 20 10
0012' 0B
0013' ED 43 0000*
0017' 21 0001*
001A' 11 0000*
001D' 01 007F
0020' ED B0

00330
00340 A, (HL)
00350 Z, MEM008
00360 P, MEM010
00370 A
00380 100
00390 N, MEM020U
00390 A, 99
00400 (HL), A
00410 INC IX
00420 INC IX
00430 POP AF
00440 JR MEM005
00450 MEM090: POP AF
00460 POP IX
00470 POP HL
00480 POP AF
00490 RET
00500 END
00100 TITLE HISTSR
00110 ENTRY
;*****
; * NXTHIS, HISTRY, MSG1, LINE13, DSPMES
; *
; * HISTORY UPDATE
; *
; * UPDATES 128 PRINT POSITIONS OF HISTORY MESSAGE
; *
; * ENTRY: (A)='L', 'W', OR 'D' FOR LOSE, WIN, DRAW
; *
; * EXIT: BUFFER UPDATED
; *
; * ALL REGISTERS SAVED
; *
;*****
00200
00210
00220
00230
00240 LD HL, (NXTHIS)
00250 INC HL
00260 LD BC, HISTRY+128
00270 LD BC, HISTRY+128
00280 OR A
00290 SBC HL, BC
00300 JR NZ, HIS010
00310 DEC BC
00320 LD HL, (NXTHIS), BC
00330 LD HL, HISTRY+1
00340 LD DE, HISTRY
00350 LD BC, 127
00360 LDIR

HISTUP: PUSH BC
PUSH OE
PUSH HL
LD HL, (NXTHIS)
INC HL
LD BC, HISTRY+128
OR A
SBC HL, BC
JR NZ, HIS010
DEC BC
LD HL, (NXTHIS), BC
LD HL, HISTRY+1
LD DE, HISTRY
LD BC, 127
LDIR

; ADJUST COUNT
; GO IF ZERO
; GO IF POSITIVE
; COUNT OF 0
; TEST FOR LT 100
; GO IF LT 100
; MAX COUNT
; STORE COUNT
; BUMP MOVE VARIABLE PNTR

; CONTINUE
; RESTORE REGISTERS

; RETURN

```

```

0022' 2A 0000#
0025' 77
0026' 21 0000#
0029' 01 0000#
002C' CD 0000#
002F' E1
0030' D1
0031' C1
0032' C9

00370 HIS010: LD HL,(NXTHIS)
00380 LD (HL),A
00390 LD HL,MSG1
00400 LD EC,LIN13
00410 CALL DSPMES
00420 POP HL
00430 POP DE
00440 POP BC
00450 RET
00460 END
00100 TITLE ROTASR
00110 ROTATE
00120 ARRXL,BASBIN
00130 EXT RIND9,ROTTAB,ARRAY1,ARRAY2
00140 EXT ROTFIR
*****
00150 ;
00160 ; ROTATE SUBROUTINE
00170 ;
00180 ; FINDS ROTATION BASE 3 VALUES FOR ROTATION OF
00190 ; ARRAY1 TO 90, 180, 270 DEGREES AND MIRROR VALUES
00200 ; OF 0, 90, 180, 270 DEGREES.
00210 ; ENTRY: ARRAY1 HOLDS ORIGINAL
00220 ; EXIT: (DE)=SMALLEST BASE 3 VALUE
00230 ; 8 VALUES IN ROTTAB
00240 ; ALL REGISTERS SAVED EXCEPT DE
*****
00250 ;
00260 ROTATE: PUSH AF
00270 PUSH BC
00280 PUSH HL
00290 PUSH IX
00300 PUSH IY
00310 LD B,7
00320 LD IX,RIND9
00330 LD IY,ROTTAB
00340 INC IY
00350 INC IY
00360 ROT010: PUSH IY
00370 PUSH IY
00380 LD IX,ARRAY1
00390 LD IY,ARRAY2
00400 POP HL
00410 PUSH HL
00420 CALL ARRXL
00430 CALL DASBIN
00440 POP IY
00450 POP IY

0000' F5
0001' C5
0002' E5
0003' 0D E5
0005' FD E5
0007' 06 07
0009' DD 21 0000#
000D' FD 21 0000#
0011' FD 23
0013' FD 23
0015' FD E5
0017' DD E5
0019' DD 21 0000#
001D' FD 21 0000#
0021' E1
0022' E5
0023' CD 0000#
0026' CD 0000#
0029' DD E1
002B' FD E1

;HISTORY POINTER
;STORE STATUS
;HISTORY MESSAGE
;START OF HISTORY AREA
;DISPLAY HISTORY
;RESTORE REGISTERS

;SEVEN ROTATIONS
;ROTATE INDICES TABLE
;ROTATE VALUES PNTR
;POINT TO R90V
;VARIABLE PNTR
;INDICES PNTR
;ORIGINAL
;NEW
;LIST OF INDICES
;SAVE
;TRANSLATE TO #
;LIST POINTER
;VARIABLE PNTR

```

375


```

00019, 0D 86 05 A, (IX+5)
00020, 00 30 10 LO (IX+1), A
00021, 00 1C 01 A, (IX-6)
00022, 00 86 07 A, (IX+7)
00023, 00 86 08 A, (IX+8)
00024, 00 86 09 A, (IX+9)
00025, 00 86 0A A, (IX+10)
00026, 00 86 0B A, (IX+11)
00027, 00 86 0C A, (IX+12)
00028, 00 86 0D A, (IX+13)
00029, 00 86 0E A, (IX+14)
00030, 00 86 0F A, (IX+15)
00031, 00 86 10 A, (IX+16)
00032, 00 86 11 A, (IX+17)
00033, 00 86 12 A, (IX+18)
00034, 00 86 13 A, (IX+19)
00035, 00 86 14 A, (IX+20)
00036, 00 86 15 A, (IX+21)
00037, 00 86 16 A, (IX+22)
00038, 00 86 17 A, (IX+23)
00039, 00 86 18 A, (IX+24)
00040, 00 86 19 A, (IX+25)
00041, 00 86 1A A, (IX+26)
00042, 00 86 1B A, (IX+27)
00043, 00 86 1C A, (IX+28)
00044, 00 86 1D A, (IX+29)
00045, 00 86 1E A, (IX+30)
00046, 00 86 1F A, (IX+31)
00047, 00 86 20 A, (IX+32)
00048, 00 86 21 A, (IX+33)
00049, 00 86 22 A, (IX+34)
00050, 00 86 23 A, (IX+35)
00051, 00 86 24 A, (IX+36)
00052, 00 86 25 A, (IX+37)
00053, 00 86 26 A, (IX+38)
00054, 00 86 27 A, (IX+39)
00055, 00 86 28 A, (IX+40)
00056, 00 86 29 A, (IX+41)
00057, 00 86 2A A, (IX+42)
00058, 00 86 2B A, (IX+43)
00059, 00 86 2C A, (IX+44)
00060, 00 86 2D A, (IX+45)
00061, 00 86 2E A, (IX+46)
00062, 00 86 2F A, (IX+47)
00063, 00 86 30 A, (IX+48)
00064, 00 86 31 A, (IX+49)
00065, 00 86 32 A, (IX+50)
00066, 00 86 33 A, (IX+51)
00067, 00 86 34 A, (IX+52)
00068, 00 86 35 A, (IX+53)
00069, 00 86 36 A, (IX+54)
00070, 00 86 37 A, (IX+55)
00071, 00 86 38 A, (IX+56)
00072, 00 86 39 A, (IX+57)
00073, 00 86 3A A, (IX+58)
00074, 00 86 3B A, (IX+59)
00075, 00 86 3C A, (IX+60)
00076, 00 86 3D A, (IX+61)
00077, 00 86 3E A, (IX+62)
00078, 00 86 3F A, (IX+63)
00079, 00 86 40 A, (IX+64)
00080, 00 86 41 A, (IX+65)
00081, 00 86 42 A, (IX+66)
00082, 00 86 43 A, (IX+67)
00083, 00 86 44 A, (IX+68)
00084, 00 86 45 A, (IX+69)
00085, 00 86 46 A, (IX+70)
00086, 00 86 47 A, (IX+71)
00087, 00 86 48 A, (IX+72)
00088, 00 86 49 A, (IX+73)
00089, 00 86 4A A, (IX+74)
00090, 00 86 4B A, (IX+75)
00091, 00 86 4C A, (IX+76)
00092, 00 86 4D A, (IX+77)
00093, 00 86 4E A, (IX+78)
00094, 00 86 4F A, (IX+79)
00095, 00 86 50 A, (IX+80)
00096, 00 86 51 A, (IX+81)
00097, 00 86 52 A, (IX+82)
00098, 00 86 53 A, (IX+83)
00099, 00 86 54 A, (IX+84)
00100, 00 86 55 A, (IX+85)
00101, 00 86 56 A, (IX+86)
00102, 00 86 57 A, (IX+87)
00103, 00 86 58 A, (IX+88)
00104, 00 86 59 A, (IX+89)
00105, 00 86 5A A, (IX+90)
00106, 00 86 5B A, (IX+91)
00107, 00 86 5C A, (IX+92)
00108, 00 86 5D A, (IX+93)
00109, 00 86 5E A, (IX+94)
00110, 00 86 5F A, (IX+95)
00111, 00 86 60 A, (IX+96)
00112, 00 86 61 A, (IX+97)
00113, 00 86 62 A, (IX+98)
00114, 00 86 63 A, (IX+99)
00115, 00 86 64 A, (IX+100)
00116, 00 86 65 A, (IX+101)
00117, 00 86 66 A, (IX+102)
00118, 00 86 67 A, (IX+103)
00119, 00 86 68 A, (IX+104)
00120, 00 86 69 A, (IX+105)
00121, 00 86 6A A, (IX+106)
00122, 00 86 6B A, (IX+107)
00123, 00 86 6C A, (IX+108)
00124, 00 86 6D A, (IX+109)
00125, 00 86 6E A, (IX+110)
00126, 00 86 6F A, (IX+111)
00127, 00 86 70 A, (IX+112)
00128, 00 86 71 A, (IX+113)
00129, 00 86 72 A, (IX+114)
00130, 00 86 73 A, (IX+115)
00131, 00 86 74 A, (IX+116)
00132, 00 86 75 A, (IX+117)
00133, 00 86 76 A, (IX+118)
00134, 00 86 77 A, (IX+119)
00135, 00 86 78 A, (IX+120)
00136, 00 86 79 A, (IX+121)
00137, 00 86 7A A, (IX+122)
00138, 00 86 7B A, (IX+123)
00139, 00 86 7C A, (IX+124)
00140, 00 86 7D A, (IX+125)
00141, 00 86 7E A, (IX+126)
00142, 00 86 7F A, (IX+127)
00143, 00 86 80 A, (IX+128)
00144, 00 86 81 A, (IX+129)
00145, 00 86 82 A, (IX+130)
00146, 00 86 83 A, (IX+131)
00147, 00 86 84 A, (IX+132)
00148, 00 86 85 A, (IX+133)
00149, 00 86 86 A, (IX+134)
00150, 00 86 87 A, (IX+135)
00151, 00 86 88 A, (IX+136)
00152, 00 86 89 A, (IX+137)
00153, 00 86 8A A, (IX+138)
00154, 00 86 8B A, (IX+139)
00155, 00 86 8C A, (IX+140)
00156, 00 86 8D A, (IX+141)
00157, 00 86 8E A, (IX+142)
00158, 00 86 8F A, (IX+143)
00159, 00 86 90 A, (IX+144)
00160, 00 86 91 A, (IX+145)
00161, 00 86 92 A, (IX+146)
00162, 00 86 93 A, (IX+147)
00163, 00 86 94 A, (IX+148)
00164, 00 86 95 A, (IX+149)
00165, 00 86 96 A, (IX+150)
00166, 00 86 97 A, (IX+151)
00167, 00 86 98 A, (IX+152)
00168, 00 86 99 A, (IX+153)
00169, 00 86 9A A, (IX+154)
00170, 00 86 9B A, (IX+155)
00171, 00 86 9C A, (IX+156)
00172, 00 86 9D A, (IX+157)
00173, 00 86 9E A, (IX+158)
00174, 00 86 9F A, (IX+159)
00175, 00 86 100 A, (IX+160)
00176, 00 86 101 A, (IX+161)
00177, 00 86 102 A, (IX+162)
00178, 00 86 103 A, (IX+163)
00179, 00 86 104 A, (IX+164)
00180, 00 86 105 A, (IX+165)
00181, 00 86 106 A, (IX+166)
00182, 00 86 107 A, (IX+167)
00183, 00 86 108 A, (IX+168)
00184, 00 86 109 A, (IX+169)
00185, 00 86 10A A, (IX+170)
00186, 00
```



```

0008' F1
0009' 77
000A' 23
000B' 10 FC
0000' 18 08

000F' F1
0010' 11 0040
0013' 77
0014' 19
0015' 10 FC
0017' E1
0018' D1
0019' C1
001A' C9

00300 POP AF
00310 LD (HL),A
00320 INC HL
00330 DJNZ DRA050
00340 JR DRA090
00350 ;VERTICAL LINE HERE
00360 DRA060: POP AF
00370 LD DE,64
00380 LD (HL),A
00390 DRA065: LD AD0
00400 DJNZ DRA065
00410 POP HL
00420 POP DE
00430 POP BC
00440 RET
00450 END
00100 TITLE SCRNSR
00110 ENTRY
00120 EXI MSGOUT,TEMP1,ARRAY1,SCARRY
00130 ;*****
00140 ; * DISPLAY SCREEN ARRAY *****
00150 ; * CONVERTS ARRAY1 TO SCREEN DISPLAY *****
00160 ; * ENTRY: NO PARAMETERS *****
00170 ; * EXIT: NO PARAMETERS *****
00180 ; * ALL REGISTERS SAVED *****
00190 ;*****
00200 ;
00210 SCRDS: PUSH AF
00220 PUSH BC
00230 PUSH DE
00240 PUSH HL
00250 PUSH IY
00260 LD HL,ARRAY1
00270 LD B,9
00280 LD IY,SCARRY
00290 LD A,(HL)
00300 OR Z,SCRO20
00310 JR 1,SCRO20
00320 CP
00330 JR NZ,SCRO15
00340 LD A,'X'
00350 JR SCRO18
00360 SCRO15: LD A,'O'
00370 SCRO18: LD (TEMP1),A
00380 PUSH HL
00390 LD HL,TEMP1

;RESTORE CHAR
;STORE GRAPHICS
;BOMP POINTER
;GO IF MORE
;DONE
;RESTORE CHARACTER
;INCREMENT
;STORE GRAPHICS
;POINT TO NEXT
;GO IF MORE
;RESTORE REGISTERS
;RETURN

```

```

0024' CD 0000*
0027' E1
0028' 23
0029' 11 000B
002C' FD 19
002E' 78
002F' FE 07
0031' 28 04
0033' FE 04
0035' 20 05
0037' 11 009F
003A' FD 19
003C' 10 D1
003E' FD E1
0040' E1
0041' 01

00400 MSGOUT
00410 HL
00420 POP
SCR020: INC
00430 LO
00440 DE,11
00450 AOD
00460 IY,0E
00470 LD
00480 A,B
00490 7
004A0 CP
004B0 JR
004C0 Z,SCR030
004D0 CP
004E0 N2,SCR040
004F0 DE,159
00500 LD
00510 IY,DE
00520 SCR040: DMZ
00530 POP
00540 HL
00550 POP
00560 DE
00570 TITLE MSGOSR
00580 ENTRY MSGOUT
00590 EXT LARGC
00600 ;*****
00610 ; LARGC MESSAGE OUTPUT ROUTINE
00620 ;
00630 ; OUTPUTS MESSAGE IN LARGE FORMAT
00640 ; ENTRY: (HL)=POINTER TO MESSAGE, MSG TERMINATED
00650 ;
00660 ; BY 0
00670 ; (IY)=POINTER TO SCREEN AREA
00680 ;
00690 ; EXIT: ALL REGISTERS SAVED
00700 ;*****
00710 ;
00720 MSGOUT: PUSH AF
00730 PUSH BC
00740 PUSH HL
00750 PUSH IY
00760 MSG010: LD A,(HL)
00770 OR A
00780 JR Z,MSG090
00790 LARGC
00800 CALL
00810 LD BC,-63
00820 ADO IY,BC
00830 INC HL
00840 JR MSG010
00850 MSG090: POP IY
00860 POP HL
00870 POP BC
00880 POP AF
00890 RET
00900 END

0000' F5
0001' C5
0002' E5
0003' FD E5
0005' 7E
0006' B7
0007' 28 0B
0009' CD 0000*
000C' 01 FEC1
000F' FD 09
0011' 23
0012' 18 F1
0014' FD E1
0016' E1
0017' C1
0018' F1
0019' C9

;DISPLAY
;RESTORE PNTR
;BUMP ARRAY PNTR
;INCREMENT
;NEXT SCREEN ADDRESS
;GET COUNT
;TEST FOR NEW ROW
;GO IF NEW ROW
;TEST FOR NEW ROW
;GO IF NOT NEW ROW
;ADJUSTMENT
;ADD ADJUSTMENT
;GO IF NOT 3 ROWS
;RESTORE REGISTERS

;SAVE REGISTERS
;GET NEXT CHARACTER
;TEST
;GO IF DONE
;DISPLAY
;DISPLACEMENT FOR 5 CP
;POINT TO NEXT CHAR POS
;POINT TO NEXT CHAR
;CONTINUE
;RESTORE REGISTERS
;RETURN

```

```

0000' 00100 TITLE LARGSR
0001' 00110 ENTRI LARGEC
0002' 00120 ;*****
0003' 00130 ; LARGC CHARACTER OISPLAY *****
0004' 00140 ; STDRS LARGC CHARACTER AT GIVEN SCREEN PDSITDN
0005' 00150 ; 8 BY 6 MATRIX USED
0006' 00160 ; ENTRY: (A)=CHARACTER TD BE STORED IN ASCII
0007' 00170 ; (IY)=SCREEN PDSITION
0008' 00180 ; EXIT: ALL REGISTERS SAVEO *****
0009' 00190 ;*****
0010' 00200 ;
0011' 00210 LARGC: PUSH AF ;SAVE REGISTERS
0012' 00220 PUSH BC
0013' 00230 PUSH HL
0014' 00240 PUSH IX
0015' 00250 LD IX,CTAB
0016' 00260 LD (IX)
0017' 00270 INC CP
0018' 00280 INC JR,NZ,LAR020
0019' 00290 PUSH IX
0020' 00300 POP HL
0021' 00310 LD BC,CTAB+1
0022' 00320 DR A
0023' 00330 SBC HL,BC
0024' 00340 PUSH HL
0025' 00350 PDP IX
0026' 00360 A00 IX,IX
0027' 00370 A00 IX,IX
0028' 00380 A00 IX,IX
0029' 00390 ADO IX,IX
0030' 00400 LD BC,D0TTAB
0031' 00410 ADO IX,BC
0032' 00420 CALL MATSR
0033' 00430 LD BC,60
0034' 00440 ADD IX,BC
0035' 00450 CALL MATSR
0036' 00460 PDP IX
0037' 00470 PDP HL
0038' 00480 PDP BC
0039' 00490 PDP AF
0040' 00500 RET
0041' 00510 MATSR: LD B,4
0042' 00520 MAT010: LD A,(IX)
0043' 00530 SET 7,A
0044' 00540 LD (IY),A
0045' 00550 INC IY
0046' 00560 INC IY
0047' 00570 INC IY
0048' 00580 INC IY
0049' 00590 INC IY
0050' 00600 INC IY
0051' 00610 INC IY
0052' 00620 INC IY
0053' 00630 INC IY
0054' 00640 INC IY
0055' 00650 INC IY
0056' 00660 INC IY
0057' 00670 INC IY
0058' 00680 INC IY
0059' 00690 INC IY
0060' 00700 INC IY
0061' 00710 INC IY
0062' 00720 INC IY
0063' 00730 INC IY
0064' 00740 INC IY
0065' 00750 INC IY
0066' 00760 INC IY
0067' 00770 INC IY
0068' 00780 INC IY
0069' 00790 INC IY
0070' 00800 INC IY
0071' 00810 INC IY
0072' 00820 INC IY
0073' 00830 INC IY
0074' 00840 INC IY
0075' 00850 INC IY
0076' 00860 INC IY
0077' 00870 INC IY
0078' 00880 INC IY
0079' 00890 INC IY
0080' 00900 INC IY
0081' 00910 INC IY
0082' 00920 INC IY
0083' 00930 INC IY
0084' 00940 INC IY
0085' 00950 INC IY
0086' 00960 INC IY
0087' 00970 INC IY
0088' 00980 INC IY
0089' 00990 INC IY
0090' 01000 INC IY
0091' 01010 INC IY
0092' 01020 INC IY
0093' 01030 INC IY
0094' 01040 INC IY
0095' 01050 INC IY
0096' 01060 INC IY
0097' 01070 INC IY
0098' 01080 INC IY
0099' 01090 INC IY
0100' 01100 INC IY

```

```

0046' 10 P2
0048' C9

0049' 41 42 43 44
0040' 45 46 47 48
0051' 49 4A 4B 4C
0055' 4D 4E 4F 50
0059' 51 52 53 54
005D' 55 56 57 58
0061' 59 5A 20 2D
0065' 3F 21

0067' 17 03 03 2B
006B' 17 03 03 2B
006F' 17 03 03 2B
0073' 37 33 33 3B
0077' 17 03 03 0B
007B' 35 30 30 38
007F' 17 03 03 29
0083' 35 30 30 1A
0087' 17 03 03 03
008B' 37 33 33 30
008F' 17 03 03 03
0093' 17 03 03 01
0097' 17 03 03 0B
009B' 35 30 30 3B
009F' 15 00 00 2A
00A3' 17 03 03 2B
00A7' 00 2B 17 00
00AB' 00 3A 35 00
00AF' 00 00 00 2B
00B3' 34 30 30 38
00B7' 15 00 20 06
00BB' 17 03 03 24
00BF' 15 00 00 00
00C3' 35 30 30 30
00C7' 1F 10 20 2F
00CB' 15 02 01 2A
00CF' 17 24 00 2A
00D3' 15 00 09 3A
00D7' 16 03 03 29
00DB' 25 30 30 1A
00DF' 17 03 03 2B
00E3' 17 03 03 03
00E7' 16 03 03 29
00EB' 25 30 38 1A

00560 ; GO IF NOT ENTIRE ROW
00570 RET ; RETURN
00580 ; TABLE OF ALLOWABLE CHARACTERS
00590 CTAB: DB 'ABCDEFGHIJKLMNOPQRSTUVWXYZ-?!',

00600 ; TABLE OF MATRICES FOR DISPLAY
00610 DOTTAB: DB 23,3,3,43,23,3,3,43 ; A
00620 DB 23,3,3,43,55,51,51,59 ; B
00630 DB 23,3,3,11,53,48,48,56 ; C
00640 DB 23,3,3,41,53,48,48,26 ; D
00650 DB 23,3,3,3,55,51,51,48 ; E
00660 DB 23,3,3,3,23,3,3,1 ; F
00670 DB 23,3,3,11,53,48,48,59 ; G
00680 DB 21,0,0,42,23,3,3,43 ; H
00690 DB 0,43,23,0,0,58,53,0 ; I
00700 DB 0,0,0,43,52,48,48,56 ; J
00710 DB 21,0,32,6,23,3,3,36 ; K
00720 DB 21,0,0,0,53,48,48,48 ; L
00730 DB 31,16,32,47,21,2,1,42 ; M
00740 DB 23,36,0,42,21,0,9,58 ; N
00750 DB 22,3,3,41,37,48,48,26 ; O
00760 DB 23,3,3,43,23,3,3,3 ; P
00770 DB 22,3,3,41,37,48,56,26 ; Q

```



```

0009' 23      00290      INC      HL      ;BUMP MESSAGE PDINTER
000A' 18 F7      JR      DSP005      ;CONTINUE
000C' E1      HL      ;RESTORE REGISTERS
000D' C1      PDP      BC
000E' F1      PDP      AF
000F' C9      RET      ;RETURN

00100      TITLE      INPUT SUBROUTINE *****
0010D      ENTRI      INPUT *****
00110      EXT      DELAY *****
00120      ;*****
00130      ;      INPUT SUBROUTINE *****
00140      ;      ; SCANS KEYBOARD FDR INPUT DF 0,1,2,3,4,5,6,7, DR *
00150      ;      ; 8 KEYS, IGNDRS ALL OTHERS. WAITS FDR KEY PRESS. *
00160      ;      ; ENTRY: ND PARAMETERS *****
00170      ;      ; EXIT: (A)=0,1,2,3,4,5,6,7, OR 8 IN BINARY *
0018D      ;      ; ALL REGISTERS EXCEPT A SAVED *****
00190      ;      ;*****
00200      ;*****
00210      ;*****
0022D      INPUT: PUSH BC      ;SAVE REGISTERS
00230      PUSH HL
00240      INF002: LD A,(3810H)
00250      OR A
00260      JR Z,INF010
00270      LD C,0FFH
00280      INC C
00290      BRCA JR NC,IMP005
00300      LD A,C
00310      JR IMP020
00320      JR IMP020
0033D      INF01D: LD A,(3820H)
00340      AND 1
00350      JR Z,IMP002
00360      LD A,8
00370      INF020: LD HL,100
00380      CALL DELAY
0039D      PDP HL
00400      PDP BC
00410      RET
00420      END
00430      TITLE      RANDSK *****
00440      ENTRI      RAND *****
00450      EXT      SEED *****
00460      ;*****
00470      ;      RANODM NUMBER ROUTINE *****
00480      ;      ; GENERATES A PSEUDO-RANDDM NUMBER FROM 0 TO 65535 *
00490      ;*****
00500      ;*****

```

```

00160      ;# ENTRY: NO PARAMETERS
00170      ;# EXIT: (BC)=RANDOM # 0-65535
00180      ;# ALL REGISTERS SAVED EXCEPT BC
00190      ;*****
00200      ;
00210      RAN0: PUSH AF
00220      PUSH DE
00230      PUSH HL
00240      LO OE,(SEEO)
00250      LO HL,(SEEO+2)
00260      LO B,7
00270      ROM010: CALL SHIFT
00280      DJNZ ROM010
00290      LO B,3
00300      ROM020: CALL SUB
00310      GJNZ ROM020
00320      LO (SEEO+2),HL
00330      ;*****WARNING***** POSSIBLE LOADER ERROR IN SOME VERSIONS
00340      ;OF ASSEMBLER. ASSEMBLER LOAO ADDRESS SHOULD BE SEEO+2 IS
00350      ;SEED!
00360      LO (SEEO+2),HL
00370      LO B,E
00380      LO C,H
00390      POP HL
00400      POP DE
00410      POP AF
00420      RET
00430      ; SHIFT SUBROUTINE
00440      SHIFT: A00 HL,HL
00450      EX OE,HL
00460      A0C HL,HL
00470      EX OE,HL
00480      RET
00490      ; SUBTRACT SEEO SUBROUTINE
00500      SUB: PUSH BC
00510      LO BC,(SEEO+2)
00520      OR A
00530      SBC
00540      EX OE,HL
00550      LO BC,(SEEO)
00560      SBC HL,BC
00570      EX OE,HL
00580      POP BC
00590      RET
00600      ENO
00610      TITLE BINSR

0000'      F5
0001'      05
0002'      E5
0003'      E0 5B 0000*
0007'      2A 0002*
000A'      06 07
000C'      C0 0025'
000F'      10 FB
0011'      06 03
0013'      C0 002B'
0016'      10 FB
0018'      E0 53 0000*

001C'      22 0002*
001F'      43
0020'      4C
0021'      E1
0022'      01
0023'      F1
0024'      C9

0025'      29
0026'      EB
0027'      E0 6A
0029'      EB
002A'      C9

002B'      C5
002C'      E0 4B 0002*
0030'      B7
0031'      E0 42
0033'      EB
0034'      ED 4B 0000*
0038'      E0 42
003A'      EB
003B'      C1
003C'      C9

```



```

00110 ENTRY BINBAS
00120 EXT DIVIDE
00130 *****
00140 ***** BINARY TO BASE 3 CONVERSION *****
00150 ***** CONVERTS A TWO-BYTE BINARY NUMBER TO BASE 3 *****
00160 ***** EQUIVALENT *****
00170 ***** ENTRI: (HL)=BINARY NUMBER *****
00180 ***** (IX)=POINTER TO ARRAY ENTRY 0 *****
00190 ***** EXIT: NUMBER IN ARRAY(0) TO ARRAY(8) *****
00200 ***** ALL REGISTERS SAVED *****
00210 *****
00220 *****
00230 BINEAS: PUSH AF ;SAVE REGISTERS
00240 PUSH BC
00250 PUSH HL
00260 PUSH IX
00270 LD BC,8
00280 ADD IX,BC
00290 LD E,3
00300 LD B,9
00310 BINO20: PUSH BC
00320 CALL DIVIDE
00330 LD A,C
00340 CP 2
00350 JR NZ,BINO30
00360 LD C,4
00370 BINO30: LD (IX),C
00380 DEC IX
00390 POP BC
00400 POP Z
00410 POP IX
00420 POP HL
00430 POP BC
00440 POP AF
00450 RET
00460 END
00470 TITLE BASBSR
00480 ENTRY BASBIN *****
00490 ***** BASE 3 TO BINARY CONVERSION *****
00500 ***** CONVERTS A BASE 3 NUMBER TO BINARY EQUIVALENT *****
00510 ***** ENTRI: (IX)=POINTER TO ARRAY ENTRY 0 *****
00520 ***** EXIT: (HL)=BINARY NUMBER, UNSIGNED *****
00530 ***** ALL REGISTERS SAVED *****
00540 *****
00550 *****
00560 *****
00570 *****
00580 *****
00590 *****
00600 *****
00610 *****
00620 *****
00630 *****
00640 *****
00650 *****
00660 *****
00670 *****
00680 *****
00690 *****
00700 *****
00710 *****
00720 *****
00730 *****
00740 *****
00750 *****
00760 *****
00770 *****
00780 *****
00790 *****
00800 *****
00810 *****
00820 *****
00830 *****
00840 *****
00850 *****
00860 *****
00870 *****
00880 *****
00890 *****
00900 *****
00910 *****
00920 *****
00930 *****
00940 *****
00950 *****
00960 *****
00970 *****
00980 *****
00990 *****

```

0000'	F5	00200	BASBIN: PUSH	AF	;SAVE REGISTERS
0001'	C5	00210	PUSH	BC	
0002'	D5	00220	PUSH	DE	
0003'	FD E5	00230	PUSH	IY	
0005'	21 0000	00240	LD	HL,0	
0008'	06 09	00250	LD	B,9	;CLEAR RESULT
000A'	E5	00260	BASO20: PUSH	HL	;LOOP COUNT
000B'	D1	00270	POP	DE	;TRANSFER RESULT TO DE
000C'	19	00280	AD	HL,DE	
000E'	FD 7E 00	00290	AD	HL,OE	;RESULT*2
0011'	FE 04	00300	LD	A,(IY)	;RESULT*3
0013'	20 02	00310	CP	4	;GET NEXT BASE 3 DIGIT
0015'	3E 02	00320	JR	NZ,BASO30	;IS IT A 4?
0017'	85	00330	LD	A,2	;GO IF NO
0018'	6F	00340	AD	A,L	;YES-BACK TO BASE 3
0019'	30 01	00350	LD	A,L	;MERGE
001B'	24	00360	JR	NC,BASO40	;NOW IN HL
001C'	FD 23	00370	INC	H	;GO IF NO CARRY
001E'	10 EA	00380	BASO40: INC	IY	;BUMP MS BYTE
0020'	FD E1	00390	DJNZ	BASO20	;POINT TO NEXT SLOT
0022'	D1	00400	POP	IY	;GO IF NOT 9
0023'	C1	00410	POP	DE	;RESTORE REGISTERS
0024'	F1	00420	POP	BC	
0026'	F1	00430	POP	AF	
0028'	C9	00440	RET		;RETURN
		00450	END		
		00100	TITLE	DIVISR	
		00110	ENTRY	DIVIDE	
		00120	*****	*****	
		00130	;	DIVIDE SUBROUTINE	
		00140	;	DIVIDE SUBROUTINE	
		00150	;	PERFORMS AN UNSIGNED DIVIDE OF 16 BITS BY 8-BIT	
		00160	;	DIVISOR.	
		00170	;	ENTRY:	
		00180	;	(HL)=DIVIDEND	
		00190	;	(E)=DIVISOR	
		00200	;	(HL)=QUOTIENT	
		00210	;	(BC)=REMAINDER	
		00220	;	ALL REGISTERS SAVED EXCEPT HL,BC	
		00230	;	*****	
		00240	DIVIDE: PUSH	DE	;SAVE REGISTERS
	D5	00250	PUSH	IX	
	DD E5	00260	PUSH	HL	;DIVIDEND TO IX
	E5	00270	POP	IX	
	DD E1	00280	LD	HL,0	;ZERO HL
	21 0000	00290	LD	D,0	;DIVISOR IN DE
	16 00		LD	B,16	;ITERATION COUNT
	000B'	06 10			

```

0000' 29      00300 OIV010: A00      HL,HL      ;SHIFT 16 MS BITS
0001' 00 29      00310 ADD      IX,IX      ;SHIFT 16 LS BITS
0010' 30 01      00320 JR      NC,OIV020 ;GO IF NO CARRY
0012' 23      00330 INC      HL          ;CARRY TO 16 MS BITS
0013' 00 23      00340 INC      IX          ;SET Q BIT
0015' B7      00350 OR       A          ;CLEAR CARRY
0016' E0 52      00360 SBC      HL,OE      ;TRY SUBTRACT
0018' 30 03      00370 JR      NC,OIV030 ;GO IF OIVIDE WENT
001A' 19      00380 A00      HL,OE      ;RESTORE
001B' 00 2B      00390 DEC      IX          ;RESET Q BIT
001C' 10 EE      00400 OIV030: OJNZ   OIV010 ;GO IF NOT 16 ITERATIONS
001F' E5      00410 PUSH     HL          ;REMAINER
0020' C1      00420 POP      BC          ;
0021' 00 E5      00430 PUSH     IX          ;QUOTIENT
0023' E1      00440 POP      HL          ;
0024' 00 E1      00450 POP      IX          ;RESTORE REGISTERS
0026' D1      00460 POP      OE          ;
0027' C9      00470 RET          ;RETURN
0028' END      00480

00100 TITLE    OELASR
00110 ENTRY    OELAY
;*****
; DELAYS 1 TO 65536 MILLISECONS.
; ENTRY: (HL)=DELAY COUNT IN MILLISECONS
; * Q=65536
; * ALL REGISTERS SAVED
;*****
00180 ;
00190 ;
00200 OELAY:   PUSH     BC
00210         PUSH     OE
00220         PUSH     HL
00230         LO       OE,-1
00240         DEC      HL
00250         DEL010: LO       B,131
00260         DEL020: DJNZ   OEL020
00270         A00      HL,OE
00280         JP       C,OEL010
00290         POP      HL
00300         POP      OE
00310         POP      BC
00320         RET
00330 ENO
00100 TITLE    FILLSR
00110 ENTRY    FILCH
;*****
00120

```

```

00130 ; *
00140 ; * FILL CHARACTER SUBROUTINE
00150 ; * ENTRY: (A)=CHARACTER
00160 ; * (05)=AREA
00170 ; * (BC)=NUMBER OF BYTES, 1-65535; 0=65536
00180 ; * ALL REGISTERS SAVED EXCEPT BC,DE
00190 ; *****
00200 ;
00210 ; FILLCH: LO (DE),A ;FILL CHARACTER
00220 INC OE ;BUMP POINTER
00230 DEC BC ;DECREMENT COUNT
00240 PUSH AF ;SAVE FILL CHARACTER
00250 LO A,B ;TEST FOR ZERO
00260 OR C
00270 JR Z,FIL010 ;GO IF DONE
00280 POP AF ;RESTORE FILL CHAR
00290 JR AF ;CONTINUE
00300 FIL010: POP AF ;RESTORE A
00310 RET ;RETURN
00320 ENO
00100 TITLE SYSTEM
00110 ENTRY MSG1,HISTY,MSG2,MSG3,MSG4,MSG5
00120 ENTRY MSG6,MSG7,MSG8,MSG9,MSG10,MSG11,MSG12
00130 ENTRY FRSTF,NXTIS,ARRAY1,ARRAY2,SEEO,RINOW
00140 ENTRY ANALTB,NORO,NORI,NOC2,NOC1,NOC2
00150 ENTRY NOO0,N001,ROTTAB,NOPTE,ROTPTR,MOVEVE
00160 ENTRY MOVEXO,NOX,NOC,NOSP,TEMP1,CRDITB,RINOT,RINOT9
00170 ENTRY RINOTR,PTABLE,STACK
00180 ; *****
00190 ; * SYSTEM MESSAGES, VARIABLES, TABLES
00200 ; *****
00210 ;
00220 ; SYSTEM MESSAGES
00230 ;
00240 MSG1: 08 , HISTORY: LAST 128 GAMES'
00250 08 ,

```

002F, 20 20 20 20
0033, 20 20 20 20
0037, 20 20 20 20
003B, 20 20 20 20
003F, 20
0040, 20 20 20 20
0044, 20 20 20 20
0048, 20 20 20 20
004C, 20 20 20 20
0050, 20 20 20 20
0054, 20 20 20 20
0058, 20 20 20 20
005C, 20 20 20 20
0060, 20 20 20 20
0064, 20 20 20 20
0068, 20 20 20 20
006C, 20 20 20 20
0070, 20 20 20 20
0074, 20 20 20 20
0078, 20 20 20 20
007C, 20 20 20 20
0080, 20 20 20 20
0084, 20 20 20 20
0088, 20 20 20 20
008C, 20 20 20 20
0090, 20 20 20 20
0094, 20 20 20 20
0098, 20 20 20 20
009C, 20 20 20 20
00A0, 20 20 20 20
00A4, 20 20 20 20
00A8, 20 20 20 20
00AC, 20 20 20 20
00B0, 20 20 20 20
00B4, 20 20 20 20
00B8, 20 20 20 20
00BC, 20 20 20 20
00C0, 00

00C1, 20 54 49 43
00C5, 2D 54 41 43
00C9, 2D 54 4F 45
00CD, 20 00
00CF, 20 20 57 41
00D3, 49 54 20 20
00D7, 4F 4E 45 20

00260 HISTRY: DB ,

00270 DB ,

00280 DB 0
00290 ; LARGE MESSAGES. 13 CHARACTERS PER LINE
00300 MSG2: DB , TIC-TAC-TOE ',0

00310 MSG3: DB , WAIT ONE ',0

```

000B' 20 00
0000' 20 20 59 4F
00E1' 55 52 20 40
00E5' 4F 56 45 20
00E9' 20 00
00EB' 20 20 54 52
00EF' 59 20 41 47
00F3' 41 49 4E 20
00F7' 20 00
00F9' 20 20 59 4F
00FO' 55 20 20 57
0101' 49 4E 21 20
0105' 20 00
0107' 20 20 4F 4E
010B' 45 20 40 4F
010F' 52 45 3F 20
0113' 20 00
0115' 20 20 20 49
0119' 20 20 57 49
0110' 4E 21 20 20
0121' 20 00
0123' 20 20 20 20
0127' 44 52 41 57
012B' 21 20 20 20
012F' 20 00
0131' 20 00
0133' 20 00
0135' 20 20 49 20
0139' 43 4F 4E 43
0130' 45 44 45 20
0141' 20 00

0143'
0144' 00
0146' 003F'
0148' 0000
0148' 0000
014A' 00
014B' 00
014B' 00
014C' 00
014C' 00
014D' 0000
014F' 00
014F' 00
0150' 1234 5678
0154' 0000

00320 MSG4: OB ' YOUR MOVE '0
00330 MSG5: OB ' TRY AGAIN '0
00340 MSG6: OB ' YOU WIN! '0
00350 MSG7: OB ' ONE MORE? '0
00360 MSG8: OB ' I WIN! '0
00370 MSG9: OB ' ORAW! '0

00380 MSG10: OB ' '0
00390 MSG11: OB '- '0
00400 MSG12: OB ' I CONCEGE '0

00410 ; VARIABLES
00420 ;
00430 ;
00440 FRSTF: OB
00450 NXTHIS: OW HISTRY-1
00460 NOPTH: OW
00470 ROTPTR: OW
00480 NOX: OW
00490 NOO: OW
00500 NOSF: OB
00510 TEMP1: OW
00520 MOVENO: OB
00530 SEEO: OW
00540 RINOW: OW
00550 ;

;0=FIRST TIME;1=NOT
;POINTER TO HISTORY LINES
;NUMBER PERMUTATION TB ENTRIES
;POINTS TO CURRENT ROTATION
;NUMBER OF XS
;NUMBER OF OS
;NUMBER OF SPACES
;TEMPORARY BUFFER
;MOVE #
;RANOOM # SEEO
;POINTER TO INVOICES

```

```

00560 ; TAELES AND ARRAYS
00570 ;
00580 ; ARRAY1: DS 9 ; ARRAY 1
00590 ARRAY2: DS 9 ; ARRAY 2
00600 ANALTB: OS 0 ; ANALYSIS TABLE
00610 MOR0: DB 0 ; ROW 0
00620 MOR1: DB 0 ; ROW 1
00630 MOR2: DB 0 ; ROW 2
00640 MOC0: DB 0 ; COLUMN 0
00650 MOC1: DB 0 ; COLUMN 1
00660 MOC2: DB 0 ; COLUMN 2
00670 MOD0: DB 0 ; DIAGONAL 0
00680 MOD1: DB 0 ; DIAGONAL 1
00690 ROTTAB: DS 0 ; ROTATION TABLE
00700 0,0,0,0,0,0,0 ; 0,90,180,270+4 MIRRORS

00710 MOVETB: DS 0 ; MOVE TABLE
00720 DW 0,0,0,0 ; MOVE 1,2,3,4 PTRS

00730 ; GRID TABLE. DEFINES TIC-TAC-TOE GRID
00740 DW 0 ; TERMINATOR
00750 GRIDTB: DS 0
00760 8CH,0,34
00770 3COOH+128+15
00780 8CH,0,34
00790 3COOH+320+15
00800 DB 8CH,0,34
00810 3COOH+512+15
00820 DB 8CH,0,34
00830 3COOH+704+15
00840 DB 0BCH,1,1
00850 3COOH+128+15
00860 DB 0BCH,1,1
00870 3COOH+128+26
00880 DB 0BCH,1,1
00890 3COOH+128+37
00900 DB 0BCH,1,1
00910 3COOH+128+48
00920 DB 0BFH,1,9
00930 3COOH+192+15
00940 DB 0BFH,1,9
00950 3COOH+192+26
00960 DB 0BFH,1,9
00970 3COOH+192+37

```

01C1'	BF 01 09	00980	DB	0BFH,1,9
01C4'	3CF0	00990	DW	3C00H+192+H8
01C6'	8F 01 01	01000	DB	8FH,1,1
01C9'	3ECF	01010	DW	3C00H+704+15
01CB'	8F 01 01	01020	DW	8FH,1,1
01CE'	3EDA	01030	DW	3C00H+704+26
01D0'	8F 01 01	01040	DB	8FH,1,1
01D3'	3EES	01050	DW	3C00H+704+37
01D5'	8F 01 01	01060	DB	8FH,1,1
01D8'	3EFO	01070	DW	3C00H+704+H8
01DA'	FF	01080	DB	0FFH
; ROTATE INDICES TABLE. HOLDS INDICES FOR ROTATIONS				
01DB'	00 01 02 03	01090	DB	0,1,2,3,4,5,6,7,8 ;0 DEG
01DF'	04 05 06 07	01100	RINDT:	DB
01E3'	08	01110	RINDT9:	DB
01E4'	02 05 08 01			2,5,8,1,4,7,0,3,6 ;90 DEG
01E8'	04 07 00 03	01120	.	DB
01EC'	06			8,7,6,5,4,3,2,1,0 ;180
01ED'	08 07 06 05	01130	DB	6,3,0,7,4,1,8,5,2 ;270
01F1'	04 03 02 01			
01F5'	00	01140	DB	2,1,0,5,4,3,8,7,6 ;MIRROR 0 DEG
01F6'	06 03 00 07	01150	DB	0,3,6,1,4,7,2,5,8 ;MIRROR 90
01FA'	04 01 08 05	01160	DB	6,7,8,3,4,5,0,1,2 ;MIRROR 180
01FE'	02	01170	DB	8,5,2,7,4,1,6,3,0 ;MIRROR 270
01FF'	02 01 00 05	01180	RINDT:	DB
0203'	04 03 08 07			0,1,2,3,4,5,6,7,8 ;INVERSE
0207'	06	01190	DB	6,3,0,7,4,1,8,5,2
0208'	00 03 06 01	01200	DB	8,7,6,5,4,3,2,1,0
020C'	04 07 02 05	01210	DB	2,5,8,1,4,7,0,3,6
0210'	08			
0211'	06 07 08 03			
0215'	04 05 00 01			
0219'	02			
021A'	08 05 02 07			
021E'	04 01 06 03			
0222'	00			
0223'	00 01 02 03			
0227'	04 05 06 07			
022B'	08			
022C'	06 03 00 07			
0230'	04 01 08 05			
0234'	02			
0235'	08 07 06 05			
0239'	04 03 02 01			
023D'	00			
023E'	02 05 08 01			


```

0242' 04 07 00 03
0246' 06
0247' 02 01 00 05
0248' 04 03 08 07
024F' 06
0250' 00 03 06 01
0254' 04 07 02 05
0258' 08
0259' 06 07 08 03
0250' 04 05 00 01
0261' 02
0262' 08 05 02 07
0266' 04 01 06 03
026A' 00

026B'
02CF'
02CF'
02CF'

01220 0B 2,1,0,5,4,3,8,7,6
01230 0B 0,3,6,1,4,7,2,5,8
01240 0B 6,7,8,3,4,5,0,1,2
01250 0B 8,5,2,7,4,1,6,3,0

; STACK AREA
01260 OS 100 ;100 BYTES
01270 OS $ ;TOP OF STACK
01280
01290 ; PERMUTATION TABLE. HOLOS BASE 3 NUMBER REPRESENTING CONFIGURATION!
01300 PTABLE: OS 0
01310 ENO

```

Figure 14-18.
Tic-Tac-Toe
Listings

The last module of the listings is the SYSTEM module. SYSTEM contains all messages, variables, and tables for the program, since we don't know its length beforehand.

The last, and most important, table is PTABLE, the permutation table we've been talking about. Since this is generated completely by the program, it's simply defined as the last location of the program.

Moving backwards, we find RINDT. This is a "Rotation Indices" table used by subroutine ROTATE. It contains the indices for converting from the "standard" array into rotations and mirror images. The last half of the table contains the indices from reconversion.

Hints and Kinks 14-4

Retranslation

We've talked about translating from one array to another one representing a rotation or mirror image. Unfortunately, retranslation back from the second to the first is not straightforward. To convert to a 90 degree rotation, for example, we have -

						OLD INDEX	NEW INDEX
0	1	2	6	3	0	0	2
3	4	5	7	4	1	1	5
6	7	8	8	5	2	2	0
						3	4
						4	1
						5	7
						6	3
						7	8
						8	6

To take the new array and reconvert back into the old array after processing, we'd have -

						NEW INDEX	OLD INDEX
0	1	2	2	5	8	0	2
3	4	5	1	4	7	1	5
6	7	8	0	3	6	2	0
						3	4
						4	1
						5	7
						6	3
						7	8
						8	6

The translation indices on reconversion bear some relationship to the conversion indices (each is 8-conversion index), but this relationship doesn't hold for other reconversions!

It's probably simplest to establish another table of reconversion indices, which we have done in the last half of RINDT.

There must be an easier way to implement this program! (Maybe you'll be the one to discover the way.)

The GRIDTB is a table of values used by the DRAWL subroutine to draw the tic-tac-toe "grid." Each five bytes defines one line.

The MOVETB table holds the address of each space cell used in a computer move. Each of the four entries may point to a space cell in the PTABLE so that it may be "adjusted" at the end of each game.

ANALTB is the "Analysis Table." It holds a count for each row, column, and diagonal of the current array. We'll discuss this under the ANALSR.

The ROTTAB table is the "Rotation Table." It holds the base three values for each of the eight rotations and mirror images of the current array. As these may be up to 19683, the entries are eight "words" of 16 bits.

There are 12 system messages at the beginning of SYSTEM. All of these are terminated by a 0.

The variables are described in the comments field of each variable, and we'll discuss them in the subroutine description.

Program Description

Here, as in MORG, we'll use a "bottom-up" approach to describe the system modules, starting from the least complex.

Fill Character Subroutine (FILLCH)

The FILLCH subroutine fills a given character into any area of memory. It decrements the byte count in BC down to zero and fills memory by using DE as a pointer register pair.

Delay Subroutine DELAY

The Delay subroutine delays from 1 through 65536 milliseconds, depending upon the count in the HL register pair. It is used only for large delays for display of system messages in tic-tac-toe.

Divide Subroutine (DIVIDE)

The Divide subroutine implements a divide of a 16-bit number in HL by an 8-bit number in E. The quotient is in HL and the remainder in BC on exit. DIVIDE is used by BINBAS for binary to base three conversion, although it is a general-purpose divide.

Base Three to Binary Subroutine (BASBIN)

BASBIN is a specialized subroutine for tic-tac-toe. It converts an array representing a tic-tac-toe configuration to a base three number. The array is either ARRAY1 or ARRAY2 in the system; each is a nine-byte table, holding a 0 for a space, 1 for an X, and 4 for an O. At the end of the conversion, HL holds the base three number in binary, 0 through 19683. The array is organized as shown in Figure 14-19.

ARRAY +0	VALUE FOR SPACE 0	VALUE = 0 FOR SPACE = 1 FOR X = 4 FOR O
+1	VALUE FOR SPACE 1	
+2	VALUE FOR SPACE 2	
+3	VALUE FOR SPACE 3	
+4	VALUE FOR SPACE 4	
+5	VALUE FOR SPACE 5	
+6	VALUE FOR SPACE 6	
+7	VALUE FOR SPACE 7	
+8	VALUE FOR SPACE 8	

0	1	2
3	4	5
6	7	8

Figure 14-19. ARRAY1/2
Format

Binary to Base Three Subroutine (BINBAS)

The Binary to Base Three subroutine does the opposite of BASBIN — it converts a binary number of 0 through 19683 to nine base-three digits of 0, 1, or 2. It then converts the 2

to a 4 and stores the nine digits in a given array (ARRAY1 or ARRAY2). The binary number is the "current value" of the configuration.

Random Number Routine (RAND)

RAND generates a pseudo-random number from a "seed" value. Its operation is virtually identical to RAND in the MORG program. On exit, BC contains a random number from 0 through 65535. This number is used to "shuffle" the "balls," or space cells, for the current configuration, to choose one of the space cells for a computer move.

Input Subroutine (INPUSR)

The Input subroutine detects a key press of 0 through 9 and ignores all others. Tic-tac-toe uses only these keys for user input to define the play. A debounce of 100 milliseconds is performed by calling subroutine DELAY.

Display Message Subroutine (DSPMES)

DSPMES is a subroutine to display a given string of ASCII characters on the screen. On entry, HL points to the string, and BC points to the screen position. The routine picks up a character from the HL pointer and then outputs it by using BC as a pointer. The string is terminated on a zero (null) character. DSPMES is only called by HISTUP to output the "history" message along the bottom of the screen.

Large Character Display Subroutine (LARGECH)

This subroutine is a general-purpose subroutine to output "large" characters of 8 by 8 pixels to the screen. The subroutine is called with A containing the character to be output in ASCII and IY pointing to the upper left-hand pixel of the 8 by 8 block to be used.

The characters A through Z, space, "-", "?", and "!" may be output. They are defined in the DOTTAB table of LARGECH as an 8 by 8 dot matrix, reading from left to right and top to bottom.

Large Message Output Subroutine (MSGOUT)

MSGOUT is similar to DSPMES except that it calls LARGECH to output a large character message. On entry, HL points

to the message and IY points to the start of the screen area for which the message is intended. IY will point to the upper left-hand pixel of the first character position of the message.

MSGOUT calls LARGEK until a terminating zero (null) character is detected in the message. Most tic-tac-toe messages are large character messages, except for the "history" message.

Display Screen Array Subroutine (SCRNDS)

SCRNDS is a specialized subroutine to convert an array (ARRAY1 or ARRAY2) to a tic-tac-toe grid with Xs and Os. The "grid" of the tic-tac-toe pattern is never rewritten during the program. SCRNDK, therefore, only fills in Xs and Os in the nine tic-tac-toe positions.

SCRNDS scans through the array from top to bottom and tests to see whether an X or O is present. If a blank is found, nothing is output for the position. If an X or O is found, SCRNDK calls MSGOUT with a "dummy" string consisting of an X or O followed by a zero (null). It puts this dummy string in the TEMP1 variable, a 16-bit variable allocated for this purpose.

Draw Line Subroutine (DRAWL)

The DRAWL subroutine draws either a vertical or horizontal line on the screen. The line is drawn through **character positions** rather than **pixel** positions. This makes the screen addressing a trivial rather than complex task.

On entry A contains the graphics character to be used. C contains a zero if a horizontal line is to be drawn, or a non-zero if a vertical line is to be drawn. HL contains the screen start position. B contains the number of character positions to be used.

DRAWL first decides whether a vertical or horizontal line is to be drawn. A branch is made to the proper code for each. If a horizontal line is to be drawn, A is stored indirect to

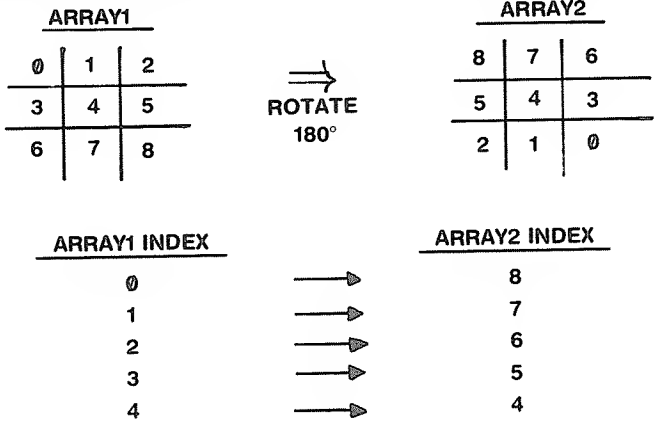
HL, HL is incremented, and B is decremented down to zero. If a vertical line is to be drawn, A is stored, HL is incremented by 64, and B is decremented down to zero. Note that vertical lines must start at the top position and horizontal lines must start at the left.

DRAWL is called by MAIN1. MAIN1 uses the GRIDTB table in the system to draw a complete set of lines representing the tic-tac-toe grid. The GRIDTB table is set up so that each entry is five bytes long, corresponding to values to be put into A, B, C, and HL. It calls DRAWL repeatedly until it has used every GRIDTB entry.

Array Translator Subroutine (ARRXLA)

The Array Translator Subroutine performs a rotation or mirror image translation from one array to the next. It is used in conjunction with BASBIN to find the new array and then to find the new base 3 value represented from the new array.

As an example, suppose that a rotation of 180 degrees is to be done on the current array. ARRXLA is entered with IX pointing to ARRAY1 in SYSTEM. IY points to ARRAY2 in SYSTEM. HL points to the list of 9 indices for the 180 degree rotation in SYSTEM at 1ED'. The nine indices of ARRAY1 are converted to the nine indices of ARRAY2 as shown in Figure 14-20. When the conversion is done, ARRAY2 holds a tic-tac-toe configuration rotated 180 degrees from ARRAY1. A BASBIN can now be done on ARRAY2.



5	→	3
6	→	2
7	→	1
8	→	0

Figure 14-20. Array Translation

ARRXLA is chiefly called by the ROTATE subroutine, which performs the seven rotations for finding the smallest value initially for the PTABLE, and then, later on, rotates the current tic-tac-toe array to find the smallest value for the PTABLE search.

Analyze Array (ANALAR)

ANALAR analyzes the current array to find eight **hash** values for the rows, columns, and diagonals of the given array. Although we've been discussing a base three value that represents the current configuration, the values actually held in the current (ARRAY1) or working (ARRAY2) arrays are actually 0 for space, 1 for X, and 4 for O. The reason for **these** values is that a simple add can give us unique values for the numbers of Xs, Os, and spaces in each row, column, or diagonal.

Hints and Kinks 14-5— Hash Values

A hashing technique uses an approach something like this: Is there a way to convert all possible actions to a series of unique numeric values that can then be used efficiently in processing?

An example from an assembler: Some assemblers add the ASCII characters in a symbol together to get a hash total. It turns out that this is (relatively) unique. The sum of "NAME", for example, is different from "START". This one-byte hash value can then be used in a more efficient search of the symbol table than a six- or seven-byte string comparison; this speeds up symbol table searches and reduces assembly time.

If the row, column, or diagonal has three spaces, the hash value will be 0. If it has two spaces and an X, the hash will be 1. Two spaces and an O yields a 4. One space and two Xs gives a 2. One space and two Os gives an 8. One space and an X and O gives a 5. No spaces and three Xs gives a 3. No spaces and two Xs and one O gives 6. No spaces and one X and two Os gives 9. No spaces and three Os gives 12. All of these values are **unique** and represent only one defined configuration. A test can, therefore, be easily made for "two Xs and a space," or "three Os." This greatly simplifies testing for tic-tac-toe conditions.

At the end of ANALAR, the eight hash values are saved in the ANALTB (Analyze Table) of SYSTEM.

Number Subroutine (NUMBER)

The NUMBER subroutine also helps to analyze an array. It counts the number of Xs, Os, and spaces in the given array (ARRAY1 or ARRAY2). The number of each is put in variables NOX, NOO, or NOSP in SYSTEM. NUMBER is called by MAIN1 to analyze the configuration for PTABLE.

Rotate Subroutine (ROTATE)

ROTATE calls ARRXL8 eight times to perform the rotations and mirror image translations of the current array. After each call, the translated array is converted to a numeric value by BASBIN. This numeric value is then put in ROTTAB in SYSTEM. At the end of ROTATE, the eight base three values are in ROTTAB and the DE register holds the lowest value. This lowest value is then used either to establish the PTABLE entry (MAIN1) or for the search of PTABLE.

History Update (HISTUP)

HISTUP is called at the end of the game with a L (lose), W (win), D (draw), or C (concede) in the A register. This code is then put in the last position of the history buffer, and the entire history message is then output to show the history of the last 128 games. If 128 have been played,

HISTUP “slides” the last 127 games into the first 127 game positions by an LDIR and then stores the current game code in the 128th position.

Memory Subroutine (MEMORY)

MEMORY implements the “reward”/“punishment” action at the end of each game. It is entered with a -1 in A for a lost game, a 3 for a win, and a 1 for a draw.

MEMORY then uses the MOVETB, which has a record of the space cell addresses, to add or subtract counts from the space cells that were used in the games. A comparison is made for increments over 99. If the count is greater than 99 (on a win or draw), the space cell is set to 99. A comparison is also made for decrements below 0. If below 0 (on a loss), the space cell is set to 0.

Main Driver Modules

There are four main driver modules, named MAIN1, MAIN2, MAIN3, and MAIN4. MAIN1 is used to perform “first time” actions, primarily to build up the PTABLE. Since this action takes several minutes, it is only done one time on each load. MAIN2 through MAIN4 are used to implement the actual game-playing.

MAIN1

MAIN1 first clears the screen with graphics 80H characters (FILLCH) and then displays the history message (DSPMES). Initially the history message will be filled with blanks, since no games have been played. DRAWL is then used to draw the grid.

Next, the Move Table MOVETB is cleared. An entry of 0 is used as a **terminator** since 0 is not a valid address for a PTABLE entry.

Now the First Time Flag (FRSTF) is tested to see if this is the first time through the program. If not, the program goes on to MAIN2 actions. If it is the first time, the history line is filled with blanks (to handle any restart of an already executed program), and a WAIT ONE message is displayed on the screen (MSGOUT).

Now the Permutation Table (PTABLE) is generated. The algorithm used is identical to that described above under "Generation of a Permutation Table." A count is incremented from 000000000 through 222222222 in binary form. BINBAS is used to find the equivalent tic-tac-toe array. NUMBER is called to count the number of Xs, Os, and spaces. ANALAR is called to analyze the eight rows, columns, and diagonals.

If the array passes all of the tests, ROTATE is then called to find the lowest value of the eight possible rotations and mirror images. If the current array is the one producing the lowest value, a new PTABLE entry is made. The number of spaces in the array is then counted and stored in the next byte. Space cells are allocated dependent upon the number of spaces. The number of spaces is then divided by 2 to produce the initial count to be put in each space cell. The quotient of this divide is conveniently equal to 4,3,2, or 1 per the algorithm.

If an entry in PTABLE is made, a dash is alternately blinked on and off at the end of the WAIT ONE message to show processing activity.

MAIN2

MAIN2 is entered at the end of PTABLE computations for the first time or reentered for the main loop in playing games. The main loop is ART070, entered from MAIN1 for each new game. Location MAINLP is reentered for each new move of a game.

If this is a start of a new game, ART070 outputs the title message and then clears the main array, ARRAY1 (FILLCH). MAINLP is then entered. When MAINLP is entered, the game has either just started or has been going on for a number of moves. In either case the action is the same. A call is made to ANALAR to analyze ARRAY1.

If two Xs exist, the computer finishes the game in the code at ART102. To do this, however, it must try an X in spaces until it finds one that produces an analyzed value of 3 (three Xs). It laboriously calls ANALAR after each try.

When the proper place for the X is found, it outputs an I WIN message and goes to the "end action" code at MAINEA in MAIN4.

If two Xs are not present, the program then tests the Move Number variable MOVENO for 5. MOVENO represents the number of times the computer has played and is incremented after each computer play. If this is the last (5th) move, the computer plays in the only space, and then tests for a win. If there's a win, it goes to the action in ART106. If there's a draw (there cannot be a loss), it outputs a DRAW! message and goes to the end action code in MAINEA.

If ART114 is entered, none of the above applies. ROTATE is now called to find the eight rotation values. ROTATE returns the smallest value in DE. This is the PTABLE value that will be used for the search. A search is now made (ART120). The PTABLE value must be found if PTABLE has been constructed properly. After the value is found, MAIN3 is entered.

MAIN3

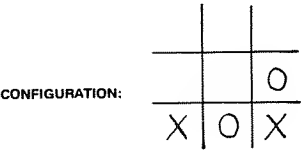
With the PTABLE entry found, the program now must handle two arrays. The first, in ARRAY1, is the present tic-tac-toe array. The second, in ARRAY2, represents the translated array from PTABLE. If the PTABLE value was the same as the present array, both arrays will be identical (0 degrees rotation), otherwise ARRAY2 will hold a rotated or mirror image array.

ARRXLA is called to convert the present array to the rotation or mirror image in ARRAY2. Then ARRAY2 is analyzed by ANALAR. If two Os are found, the computer "blocks" the move by putting in an X in the proper space. The code at ART102 is used to find the space which produces the proper block, and the code at ART106 finds the proper space cell for the blocking move. After the block, a JP is made to ART172 for further processing.

If a blocking move is not possible, the code at ART125 is performed. This code finds the total count of all space cells

associated with the permutation. If this total count is zero, this path is hopeless and will be deleted. The MOVETB is used to find the last move, and the space cell for the last move is zeroed. An I CONCEDE message is then output, and the end action at MAINEA is performed. This concession action will rarely happen.

If the total count is non-zero, a random number **less than or equal to** the total count is found by calling RAND. The space cells are totaled until the total is equal to or greater than the random number. This action is analogous to stirring the balls in the box and picking one! (See Figure 14-21.)



PTABLE ENTRY:

46H (000 002 121)	CONFIGURATION IN BASE 3
5	# OF SPACE CELLS
10	COUNT FOR SPACE 0
2	COUNT FOR SPACE 1
7	COUNT FOR SPACE 2
1	COUNT FOR SPACE 3
1	COUNT FOR SPACE 4

PICKING A SPACE:

1. TOTAL COUNT OF SPACE CELLS = 10+2+7+1 = 21
2. FINO RANDOM # <= 21 : 12
3. O → TOTAL
4. AOO SPACE CELL 0 : TOTAL = 10, < RANOOM # OF 12
5. AOO SPACE CELL 1 : TOTAL = 12, = RANOOM # OF 12
6. SPACE CELL 1 WILL BE USED FOR MOVE, CAUSING X TO BE PUT INTO SPACE 1

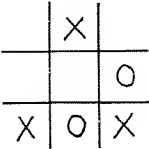


Figure 14-21. Using RAND To Pick A Space

Now an X is stored in the array (ARRAY2), and a record is made of the space cell move in MOVETB. We worked with ARRAY2 and then converted it back to ARRAY1 by a call to ARRXLA. RINDW holds the pointer to the "reverse indices." SCRNS is then called to display ARRAY1, and MAIN4 is entered.

MAIN4

MAIN4 is used to get the human input. It outputs the YOUR MOVE message and waits for user input from INPUT. A check is made of the validity of the number of the square chosen and a TRY AGAIN message output.

When the human inputs a valid move, an O is stored in ARRAY1 and SCRNS is called to display the array. ANALAR is then called (ART185) to analyze the new array. Only a human win can occur at this point. If the human wins, a YOU WIN message is output and the end action at MAINEA is performed. If no win occurred, the next computer move must be done and MAINLP in MAIN2 is reentered.

The end action at MAINEA is entered at the completion of every game. Before this point the HISTUP (History Update) and MEMORY routines have been called to update the history message and "reward" or "punish" the PTABLE space cell. MAINEA outputs a ANOTHER? message and waits for a key press. On the key press, ARTIP in MAIN1 is reentered for a new game.

Using This Program

As in the case of MORG, this program was designed and coded to let you see a significant chunk of presumably worthwhile code. If you'd like to experiment with the program, you may enter the machine code by using Disk DEBUG or T-BUG. Figure 14-22 gives the machine code after the program load. There are several thousand bytes, but a fast typist could enter them in an hour. **Checkpoint** by saving partial results! You can do this by dumping to disk (by DUMP) or to cassette (P command in T-BUG) and then reloading to take up where you left off.

8000	31	E7	8A	3E	80	11	00	3C	01	40	03	CD	0B	88	21	18
8010	88	01	40	3F	CD	13	87	DD	21	A2	89	DD	7E	00	FE	FF
8020	28	16	DD	4E	01	DD	46	02	DD	6E	03	DD	66	04	CD	42
8030	85	01	05	00	DD	09	18	E3	AF	11	98	89	01	08	00	CD
8040	0B	88	32	67	89	3A	5B	89	B7	C2	2C	81	3C	32	5B	89
8050	11	57	88	ED	53	5C	89	13	3E	20	01	80	00	CD	0B	88
8060	21	D9	88	FD	21	00	3C	CD	A2	85	21	B8	0B	CD	F8	87
8070	21	E7	88	FD	21	00	3C	CD	A2	85	21	E7	8A	E5	21	00
8080	00	22	5E	89	E5	01	E3	4C	B7	ED	42	CA	29	81	09	DD
8090	21	6E	89	CD	83	87	22	88	89	21	6E	89	CD	7E	84	3A
80A0	62	89	57	3A	63	89	BA	28	03	E1	18	79	FE	04	28	F9
80B0	DD	21	6E	89	CD	AC	84	06	08	21	80	89	7E	FE	03	CA
80C0	A9	80	FE	0C	CA	A9	80	FE	02	CA	A9	80	23	10	ED	CD
80D0	16	84	E1	E5	B7	ED	52	20	D0	E1	DD	E1	DD	75	00	DD
80E0	74	01	FD	2A	5E	89	FD	23	FD	22	5E	89	3A	64	89	4F
80F0	DD	71	02	DD	23	DD	23	DD	23	DD	E5	D1	06	00	DD	09
8100	DD	E5	79	CB	3F	D5	CD	0B	88	3A	5E	89	E6	01	21	49
8110	89	28	03	21	4B	89	FD	21	3C	3C	CD	A2	85	DD	E1	DD
8120	6E	FD	DD	66	FE	23	C3	84	80	E1	DD	E1	21	D9	88	FD
8130	21	00	3C	CD	A2	85	21	B8	0B	CD	F8	87	01	09	00	AF
8140	11	6E	89	CD	0B	88	21	67	89	34	DD	21	6E	89	CD	AC
8150	84	21	80	89	06	08	7E	FE	02	28	05	23	10	F8	18	3F
8160	E5	21	6E	89	7E	B7	28	03	23	18	F9	E5	3C	77	DD	21
8170	6E	89	CD	AC	84	E1	DD	E1	DD	7E	00	FE	03	26	06	AF
8180	77	DD	E5	18	E3	CD	5D	85	3E	57	CD	E3	83	3E	03	CD
8190	B1	83	21	2D	89	FD	21	00	3C	CD	A2	85	C3	9B	83	3A
81A0	67	89	FE	05	20	3A	21	6E	89	7E	B7	28	03	23	18	F9
81B0	3C	77	DD	21	6E	89	CD	AC	84	21	6E	89	06	08	7E	FE
81C0	03	28	C2	23	10	F8	CD	5D	85	3E	44	CD	E3	83	3E	01
81D0	CD	B1	83	21	3B	89	FD	21	00	3C	CD	A2	85	C3	9B	83
81E0	FD	21	6E	89	CD	AA	87	22	88	89	CD	16	84	DD	21	E7
81F0	8A	DD	6E	00	DD	66	01	B7	ED	52	28	0C	DD	4E	02	06
8200	00	03	03	03	DD	09	18	E9	DD	E5	2A	60	89	01	88	89
8210	B7	ED	42	CB	3D	CB	3C	30	02	CB	FD	E5	29	29	29	C1
8220	09	E5	01	F3	89	09	DD	21	6E	89	FD	21	77	89	CD	17
8230	85	E1	01	3B	8A	09	22	6C	89	DD	21	77	89	CD	AC	84
8240	21	80	89	06	08	7E	FE	08	28	05	23	10	F8	16	52	E5
8250	21	77	89	7E	B7	28	03	23	18	F9	E5	3C	77	DD	21	77
8260	89	CD	AC	84	E1	DD	E1	DD	7E	00	FE	09	28	06	AF	77
8270	DD	E5	18	E3	AF	77	E5	01	76	89	B7	ED	42	11	FF	FF
8280	45	21	77	89	7E	23	B7	20	01	13	10	F8	E1	3E	01	77
8290	DD	E1	DD	19	DD	23	DD	23	DD	23	DD	23	DD	E5	C3	1E
82A0	83	DD	E1	DD	E5	DD	46	02	21	00	00	DD	5E	03	16	00
82B0	19	DD	23	10	F6	7C	B5	DD	E1	DD	E5	20	25	3A	67	89
82C0	3D	07	4F	06	00	DD	21	98	89	DD	09	DD	66	00	DD	6E
82D0	01	AF	77	21	4D	89	FD	21	00	3C	CD	A2	85	3E	43	C3
82E0	93	83	CD	46	87	C5	D1	EB	B7	ED	52	30	FB	19	DD	4E
82F0	03	06	00	B7	ED	42	DD	23	28	02	30	F2	D1	DD	E5	E1
8300	01	03	00	DD	09	DD	E5	B7	ED	52	45	DD	21	77	89	DD
8310	7E	00	B7	DD	23	20	F8	10	F6	3E	01	DD	77	FF	3A	67
8320	89	3D	07	5F	16	00	21	98	89	19	D1	1B	73	23	72	2A
8330	6C	89	DD	21	77	89	FD	21	6E	89	CD	17	85	CD	5D	85
8340	21	F5	88	FD	21	00	3C	CD	A2	85	CD	23	87	4F	06	00
8350	21	6E	89	09	7E	B7	28	12	21	03	89	FD	21	00	3C	CD
8360	A2	85	21	B8	0B	CD	F8	87	18	D6	3E	04	77	CD	5D	85
8370	DD	21	6E	89	CD	AC	84	21	80	89	06	08	7E	23	FE	0C
8380	28	05	10	F8	C3	46	81	21	11	89	FD	21	00	3C	CD	A2
8390	85	3E	4C	CD	E3	83	3E	FF	CD	B1	83	21	B8	0B	CD	F8
83A0	87	21	1F	89	FD	21	00	3C	CD	A2	85	CD	23	87	C3	00
83B0	80	F5	E5	DD	E5	DD	21	98	89	DD	66	01	DD	6E	00	F5
83C0	7C	B5	28	19	F1	F5	86	CA	CD	83	F2	CE	83	AF	FE	64
83D0	FA	D5	83	3E	63	77	DD	23	DD	23	F1	18	DC	F1	DD	E1
83E0	E1	F1	C9	C5	D5	E5	2A	5C	89	23	22	5C	89	01	D8	88
83F0	B7	ED	42	20	10	0B	ED	43	5C	89	21	59	88	11	58	88
8400	01	7F	00	ED	B0	2A	5C	89	77	21	18	88	01	40	3F	CD
8410	13	87	E1	D1	C1	C9	F5	C5	E5	DD	E5	FD	E5	06	07	DD
8420	21	FC	89	FD	21	88	89	FD	23	FD	23	FD	E5	DD	E5	DD
8430	21	6E	89	FD	21	77	89	E1	E5	CD	17	85	CD	AA	87	DD
8440	E1	FD	E1	FD	75	00	FD	74	01	FD	23	FD	23	11	09	00
8450	DD	19	10	D7	06	08	DD	21	88	89	11	FF	7F	DD	66	01
8460	DD	6E	00	B7	ED	52	F2	70	84	19	54	5D	DD	22	60	89
8470	DD	23	DD	23	10	E7	FD	E1	DD	E1	E1	C1	F1	C9	F5	C5
8480	D5	E5	06	09	11	00	00	0E	00	7E	FE	01	20	01	14	FE
8490	04	20	01	1C	B7	20	01	0C	23	10	EE	7A	32	62	89	7B
84A0	32	63	89	79	32	64	89	E1	D1	C1	F1	C9	F5	FD	E5	FD
84B0	21	80	89	DD	7E	00	DD	86	01	DD	86	02	FD	77	00	DD
84C0	7E	03	DD	86	04	DD	86	05	FD	77	01	DD	7E	06	DD	86
84D0	07	DD	86	08	FD	77	02	DD	7E	00	DD	86	03	DD	86	06
84E0	FD	77	03	DD	7E	01	DD	86	04	DD	86	07	FD	77	04	DD

```

84FD 7E 02 DD 86 05 DD 86 08 FD 77 D5 DD 7E 00 DD 86
8500 04 DD 86 D8 FD 77 D6 DD 7E 02 DD 86 D4 DD 86 06
8510 FD 77 07 FD E1 F1 C9 F5 C5 D5 E5 DD E5 FD E5 06
8520 09 FD E5 5E 16 00 FD E1 FD E5 FD 19 DD 7E 00 FD
853D 77 0D 23 DD 23 10 EC FD E1 FD E1 DD E1 E1 D1 C1
8540 F1 C9 C5 D5 E5 F5 79 B7 2D 07 F1 77 23 10 FC 18
8550 08 F1 11 4D 00 77 19 1D FC E1 D1 C1 C9 F5 C5 D5
8560 E5 FD E5 21 E 89 06 09 FD 21 D3 3C 7E B7 28 15
8570 FE 01 20 04 3E 58 18 02 3E 4F 32 65 89 E5 21 65
8580 89 CD A2 85 E1 23 11 0B 00 FD 19 78 FE 07 28 04
8590 FE D4 20 D5 11 9F DD FD 19 10 D1 FD E1 E1 D1 C1
85A0 F1 C9 F5 C5 E5 FD E5 7E B7 28 0B CD BC 85 01 C1
85B0 FF FD 09 23 18 F1 FD E1 E1 C1 F1 C9 F5 C5 E5 DD
85CD E5 DD 21 D5 86 DD BE 00 DD 23 20 F9 DD E5 E1 01
85D0 06 86 B7 ED 42 E5 DD E1 DD 29 DD 29 DD 29 01 23
85E0 86 DD 09 CD F4 85 01 3C 00 FD 09 CD F4 85 DD E1
85FD E1 C1 F1 C9 06 D4 DD 7E 00 CB FF FD 77 00 DD 23
8600 FD 23 10 F2 C9 41 42 43 44 45 46 47 48 49 4A 4B
8610 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 59 5A 20
8620 2D 3F 21 17 03 03 2B 17 03 03 2B 17 03 03 2B 37
8630 33 33 3B 17 D3 03 0B 35 30 30 38 17 03 03 2B 37
8640 30 30 1A 17 03 03 03 37 33 33 30 17 03 03 03 17
8650 03 03 01 17 03 03 DB 35 30 30 3B 15 00 00 2A 17
8660 03 03 2B 00 2B 17 00 00 3A 35 00 00 00 00 2B 34
8670 3D 30 38 15 DD 20 06 17 03 D3 24 15 00 0D 00 35
8680 30 30 30 1F 10 20 2F 15 02 01 2A 17 24 00 2A 15
8690 0D 09 3A 16 D3 03 29 25 30 30 1A 17 03 D3 2B 17
86AD D3 D3 03 16 D3 03 29 25 30 38 1A 17 03 03 2B 17
86B0 03 03 25 17 03 03 03 33 33 33 3B 03 2B 17 03 00
86C0 2A 15 D0 15 00 00 2A 35 30 30 3A 15 00 DD 2A D2
86D0 24 18 01 15 28 14 2A 25 3A 35 1A 09 30 30 06 18
86E0 03 03 24 15 00 00 2A 03 2B 17 03 03 03 33 0F 3C
86F0 33 30 30 00 00 00 00 00 00 00 00 00 00 00 03
8700 03 03 D3 07 23 33 3B 00 22 11 00 00 2A 15 00 00
871D 22 11 00 F5 C5 E5 7E B7 28 05 02 03 23 18 F7 E1
8720 C1 F1 C9 C5 E5 3A 10 38 B7 28 09 0E FF 0C 0F 30
8730 FC 79 18 09 3A 20 38 E6 D1 28 EA 3E 08 21 64 00
8740 CD F8 87 E1 C1 C9 F5 D5 E5 ED 5B 68 89 2A 6A 89
8750 06 07 CD 6B 87 10 FB 06 03 CD 71 87 10 FB ED 53
8760 68 89 22 6A 89 43 4C E1 D1 F1 C9 29 EB ED 6A EB
8770 C9 C5 ED 4B 6A 89 B7 ED 42 EB ED 4B 68 89 ED 42
8780 EB C1 C9 F5 C5 E5 DD E5 01 08 0D DD 09 1E 03 06
8790 09 C5 CD D0 87 79 FE 02 20 02 0E 04 DD 71 00 DD
87A0 2B C1 10 ED DD E1 E1 C1 F1 C9 F5 C5 05 FD E5 21
87B0 00 00 06 09 E5 D1 19 19 FD 7E 00 FE 04 20 02 3E
87C0 02 85 6F 30 01 24 FD 23 10 EA FD E1 D1 C1 F1 C9
87D0 D5 DD E5 E5 DD E1 21 00 00 16 00 06 10 29 DD 29
87E0 30 01 23 DD 23 B7 ED 52 30 03 19 DD 2B 10 EE E5
87F0 C1 DD E5 E1 DD E1 D1 C9 C5 D5 E5 11 FF FF 2B 06
8800 83 10 FE 19 DA FF 87 E1 D1 C1 C9 12 13 0B F5 78
8810 B1 28 03 F1 18 F5 F1 C9 20 20 20 20 20 20 20 20
8820 2D 20 20 20 20 20 20 20 20 20 20 20 48 49 53 54
8830 4F 52 59 3A 20 4C 41 53 54 20 31 32 38 20 47 41
8840 4D 45 53 20 20 20 20 20 20 20 20 20 20 20 20 20
885D 20 20 20 20 20 20 20 20 20 20 2D 20 20 20 20 20
886D 20 2D 2D 20 20 2D 20 2D 20 2D 20 20 20 20 2D 2D
8870 20 20 20 20 2D 20 2D 20 20 20 20 2D 20 2D 2D 2D
8880 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
889D 20 20 2D 20 2D 20 20 20 20 20 20 20 20 20 20 20
88A0 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
88B0 20 20 2D 20 2D 20 20 20 20 20 20 2D 20 2D 20 20
88C0 20 20 20 20 20 20 20 20 20 20 2D 20 20 20 20 20
88D0 20 20 20 20 20 20 20 20 20 20 00 20 54 49 43 2D 54 41
88E0 43 2D 54 4F 45 20 00 20 20 57 41 49 54 20 20 4F
88F0 4E 45 20 20 00 20 20 59 4F 55 52 20 4D 4F 56 45
8900 20 20 00 20 20 54 52 59 2D 41 47 41 49 4E 20 20
8910 00 20 20 59 4F 55 20 20 57 49 4E 21 20 2D 00 20
8920 20 4F 4E 45 20 4D 4F 52 45 3F 20 20 20 20 20 20
8930 49 20 20 57 49 4E 21 20 20 20 00 20 20 20 20 44
8940 52 41 57 21 20 20 20 20 00 20 00 2D 00 20 20 49
8950 20 43 4F 4E 43 45 44 45 20 20 00 00 57 88 00 00
8960 00 00 00 00 00 00 00 00 34 12 78 56 00 00 00 00
8970 00 00 00 00 00 00 00 00 00 23 FF FF FF FF FF FF
8980 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8990 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
89A0 00 00 8C 00 22 8F 3C 8C 00 22 4F 3D 8C 00 22 0F
89B0 3E 8C 00 22 CF 3E BC 01 01 8F 3C BC D1 01 9A 3C
89C0 BC 01 01 A5 3C BC 01 01 B0 3C BF 01 09 CF 3C BF
89D0 01 09 DA 3C BF 01 09 E5 3C BF 01 09 F0 3C 8F 01

```



```

89E0 01 CF 3E 8F 01 01 DA 3E 8F 01 01 E5 3E 8F 01 01
89F0 F0 3E FF 00 01 02 03 04 05 06 07 08 02 05 08 01
8A00 04 07 00 03 06 08 07 06 05 04 03 02 01 00 06 03
8A10 00 07 04 01 08 05 02 02 01 00 05 04 03 08 07 06
8A20 00 03 06 01 04 07 02 05 08 06 07 08 03 04 05 00
8A30 01 02 08 05 02 07 04 01 06 03 00 00 01 02 03 04
8A40 05 06 07 08 06 03 00 07 04 01 08 05 02 08 07 06
8A50 05 04 03 02 01 00 02 05 08 01 04 07 00 03 06 02
8A60 01 00 05 04 03 08 07 06 00 03 06 01 04 07 02 05
8A70 08 06 07 08 03 04 05 00 01 02 08 05 02 07 04 01
8A80 06 03 00 FF FF FF FF FF FF FF FF FF FF FF FF FF
8A90 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
8AA0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
8AB0 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
8AC0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8AD0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
8AE0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

**Figure 14-22. Machine Code
to Tic-Tac-Toe**

Hints and Kinks 14-6 Checkpointing

How often should you checkpoint? I'm from the old school — I never trust computers. As a matter of fact, you may employ Barden's Law: The more you checkpoint, the less you'll need it! If you don't checkpoint, an operator error or power failure will surely wipe out several hours of work. But seriously — the TRS-80 is much less prone to losing data than minicomputers of several years ago! And reliability will continue to get better with newer hardware.

You may also key in the source for the program and use the Disk Assembler to assemble and load your own version for experimentation. This is quite a task, but it's certainly possible, especially since the program is modular. EDTASM may also be used to assemble one huge program.

APPENDIX I

Z-80 Instruction Set

A Register Operations

Complement CPL
Decimal DAA
Negate NEG

Adding/Subtracting Two 8-Bit Numbers

A and Another Register

ADC A,r SBC A,r
ADD A,r SUB A,r

A and Immediate Operand

ADC A,n SBC A,n
ADD A,n SUB A,n

A and Memory Operand

ADC A,(HL)	ADD A,(HL)	SBC (HL)	SUB (HL)
ADC A,(IX+d)	ADD A,(IX+d)	SBC (IX+d)	SUB (IX+d)
ADC A,(IY+d)	ADD A,(IY+d)	SBC (IY+d)	SUB (IY+d)

Adding/Subtracting Two 16-Bit Numbers

HL and Another Register Pair

ADC HL,ss ADD HL,ss SBC HL,ss

IX and Another Register Pair

ADD IX,pp ADD IY,rr

Bit Instructions

Test Bit

Register	BIT b,r		
Memory	BIT b,(HL)	BIT b,(IX+d)	BIT b,(IY+)

Reset Bit

Register	RES b,r		
Memory	RES b,(HL)	RES b,(IX+d)	RES b,(IY+d)

Set Bit

Register	SET b,r		
Memory	SET b,(HL)	SET b,(IX+d)	SET b,(IY+d)

Carry Flag

Complement CCF
Set SCF

Compare Two 8-Bit Operands

A and Another Register CP r
A and Immediate Operand CP n
A and Memory Operand
CP (HL) CP (IX+d) CP (IY+d)
Block Compare
CPD,CPDR,CPI,CPIR

Decrements and Increments

Single Register
DEC r INC r DEC IX DEC IY INC
Register Pair
DEC ss INC ss DEC IX DEC IY INC IX INC IY
Memory
DEC HL DEC (IX+d) DEC (IY+d)
INC (HL) INC (IX+d) INC (IY+d)

Exchanges

DE and HL EX DE,HL
Top of Stack
EX (SP),HL EX (SP),IX EX (SP),IY

Input/Output

I/O To/From A and Port
IN A,(n) OUT (n),A
I/O To/From Register and Port
IN r,(C) OUT (C),r
Block
IND,INDR,INR,INIR,OTDR,OTIR,OUTD,OUTI

Interrupts

Disable DI
Enable EI
Interrupt Mode
IM 0 IM 1 IM 2
Return From Interrupt
RETI RETN

Jumps

Unconditional
JP (HL) JP (IX) JP (IY) JP (nn) JR e
Conditional
JP cc,nn JR C,e JR NZ,e JR Z,e JR NC,e
Special Conditional
DJNZ e

Loads

A Load Memory Operand
LD A,(BC) LD A,(DE) LD A,(nn)

A and Other Registers

LD A,I LD A,R LD I,A LD R,A

Between Registers, 8-Bit

LD r,r'

Immediate 8-Bit

LD r,n

Immediate 16-Bit

LD dd,nn LD IX,nn LD IY,nn

Register Pairs From Other Register Pairs

LD SP,HL LD SP,IX LD SP,IY

From Memory, 8-Bits

LD r,(HL) LD r,(IX+d) LD r,(IY+d)

From Memory, 16-Bits

LD HL,(nn) LD IX,(nn) LD IY,(nn) LD dd,(nn)

Block

LDD,LDDR,LDI,LDIR

Logical Operations 8 Bits With A

A and Another Register

AND r OR r XOR r

A and Immediate Operand

AND n OR n XOR n

A and Memory Operand

AND (HL) OR (HL) XOR (HL)

AND (IX+d) OR (IX+d) XOR (IX+d)

AND (IY+d) OR (IY+d) XOR (IY+d)

Miscellaneous

Halt HALT

No Operation NOP

Prime/Non-Prime

Switch AF

EX AF,AF'

Switch Others

EXX

Shifts

Circular (Rotate)

A Only RLA, RLCA, RRA, RRCA

All Registers RL r RLC r RR r RRC r

Memory

RL (HL) RLC (HL) RR (HL) RRC (HL)

RL (IX+d) RLC (IX+d) RR (IX+d) RRC (IX+d)

RL (IY+d) RLC (IY+d) RR (IY+d) RRC (IY+d)

Logical

Registers SRL r

Memory SRL (HL) SRL (IX+d) SRL (IY+d)

Arithmetic

Registers SLA r SRA r

Memory

SLA (HL) SRA (HL)

SLA (IX+d) SRA (IX+d)

SLA (IY+d) SRA (IY+d)

BCD RLD RRD

Stack Operations

PUSH IX PUSH IY PUSH qq POP IX POP IY POP qq

Stores

Of A Only

LD (BC),A LD (DE),A LD (nn),A

All Registers

LD (HL),r LD (IX+d),r LD (IY+d),r

Immediate Data

LD (HL),n LD (IX+d),n LD (IY+d),n

16-Bit Registers

LD (nn),dd LD (nn),IX LD (nn),IY LD (nn),HL

Subroutine Action

Conditional CALLs CALL cc,nn

Unconditional CALLs CALL nn

Conditional Return RET cc

Unconditional Return RET

Special CALL RST p

APPENDIX II

Z-80 Operation Code Listings

Mnemonic	Format	Description	S	Z	P/V	C
ADC HL,ss	11101101 01ss1010	HL + ss + CY to HL	●	●	●	●
ADC A,r	10001 r	A + r + CY to A	●	●	●	●
ADC A,n	11001110 n	A + n + CY to A	●	●	●	●
ADC A,(HL)	10001110	A + (HL) + CY to A	●	●	●	●
ADC A,(IX + d)	11011101 10001110 d	A + (IX + d) + CY to A	●	●	●	●
ADC A,(IY + d)	11111101 10001110 d	A + (IY + d) + CY to A	●	●	●	●
ADD A,n	11000110 n	A + n to A	●	●	●	●
ADD A,r	10000 r	A + r to A	●	●	●	●
ADD A,(HL)	10000110	A + (HL) to A	●	●	●	●
ADD A,(IX + d)	11011101 10000110 d	A + (IX + d) to A	●	●	●	●
ADD A,(IY + d)	11111101 10000110 d	A + (IY + d) to A	●	●	●	●
ADD HL,ss	00ss1001	HL + ss to HL				●
ADD IX,pp	11011101 00pp1001	IX + pp to IX				●
ADD IY,rr	11111101 00rr1001	IY + rr to IY				●
AND r	10100 r	A AND r to A	●	●	●	0
AND n	11100110 n	A AND n to A	●	●	●	0
AND (HL)	10100110	A AND (HL) to A	●	●	●	0
AND (IX + d)	11011101 10100110 d	A AND (IX + d) to A	●	●	●	0
AND (IY + d)	11111101 10100110 d	A AND (IY + d) to A	●	●	●	0

BIT b,r	11001011	01 b r	Test bit b of r	0	1
BIT b,(HL)	11001011	01 b 110	Test bit b of (HL)	0	1
BIT b,(IX + d)	11011101	11001011 d	Test bit b of (IX + d)	0	1
BIT b,(IY + d)	11111101	11001011 d	Test bit b of (IY + d)	0	1
CALL cc,nn	11 c 100	n	CALL subroutine at nn if cc	0	1
CALL nn	11001101	n	Unconditionally CALL nn	0	1
CCF	00111111		Complement carry flag	0	1
CP r	10111 r		Compare A:r	0	1
CP n	11111110	n	Compare A:n	0	1
CP (HL)	10111110		Compare A:(HL)	0	1
CP (IX + d)	11011101	10111110 d	Compare A:(IX + d)	0	1
CP (IY + d)	11111101	10111110 d	Compare A:(IY + d)	0	1
CPD	11101101	10101001	Block Compare, no repeat	0	1
CPDR	11101101	10111001	Block Compare, repeat	0	1
CPI	11101101	10100001	Block Compare, no repeat	0	1
CPIR	11101101	10110001	Block Compare, repeat	0	1
CPL	00101111		Complement A (1's comple)	0	1
DAA	00100111		Decimal Adjust A	0	1
DEC r	00 r 101		Decrement r by one	0	1
DEC (HL)	00110101		Decrement (HL) by one	0	1

Mnemonic	Format	S	Z	P/V	C
----------	--------	---	---	-----	---

DEC (X + d)	11011101 00110101	d			
DEC (Y + d)	11111101 00110101	d			
DEC IX	11011101 00101011				
DEC IY	11111101 00101011				
DEC ss	00ss1011				
DI	111110011				
DJNZ e	00010000 e-2				
EI	111111011				
EX (SP), HL	11100011				
EX (SP), IX	11011101 11100011				
EX (SP), IY	11111101 11100011				
EX AF, AF'	00001000				
EX DE, HL	11101011				
EXX	11011001				
HALT	01110110				
IM 0	11101101 01000110				
IM 1	11101101 01010110				
IM 2	11101101 01011110				
IN A, (n)	11011011 n				

Description

Decrement (IX + d) by one

Decrement (IY + d) by one

Decrement IX by one

Decrement IY by one

Decrement register pair

Disable interrupts

Decrement B and JR if B \neq 0

Enable interrupts

Exchange (SP) and HL

Exchange (SP) and IX

Exchange (SP) and IY

Set prime AF active

Exchange DE and HL

Set prime B-L active

Halt

Set interrupt mode 0

Set interrupt mode 1

Set interrupt mode 2

Load A with input from n

IN $r, (C)$
 INC r
 INC (HL)
 INC (IX + d)
 INC (IY + d)
 INC IX
 INC IY
 INC ss
 IND
 INDR
 INI
 INIR
 JP (HL)
 JP (IX)
 JP (IY)
 JP cc,nn
 JP nn
 JR C,e
 JR e
 JR NC,e

11101101	01	r	000
00	r	100	
00110100			
11011101	00110100		d
11111101	00110100		d
11011101	00100011		
11111101	00100011		
00ss0011			
11101101	10101010		
11101101	10111010		
11101101	10100010		
11101101	10110010		
11101001			
11011101	11101001		
11111101	11101001		
11	c	010	n
11000011	n		n
00111000	e-2		
00011000	e-2		
00110000	e-2		

Load r with input from (C)
 Increment r by one
 Increment (HL) by one
 Increment (IX + d) by one
 Increment (IY + d) by one
 Increment IX by one
 Increment IY by one
 Increment register pair
 Block I/O input from (C)
 Block I/O input, repeat
 Block I/O input from (C)
 Block I/O input, repeat
 Unconditional jump to (HL)
 Unconditional jump to (IX)
 Unconditional jump to (IY)
 Jump to nn if cc
 Unconditional jump to nn
 Jump relative if carry
 Unconditional jump relative
 Jump relative if no carry

①
②
③
④
⑤
⑥
⑦
⑧
⑨
⑩
⑪
⑫
⑬
⑭
⑮
⑯
⑰
⑱
⑲
⑳
㉑
㉒
㉓
㉔
㉕
㉖
㉗
㉘
㉙
㉚
㉛
㉜
㉝
㉞
㉟
㊱
㊲
㊳
㊴
㊵
㊶
㊷
㊸
㊹
㊺
㊻
㊼
㊽
㊾
㊿

Mnemonic	Format	S	Z	P/V	C	Description
JR NZ,e	00100000 e-2					Jump relative if non-zero
JR Z,e	00101000 e-2					Jump relative if zero
LD A,(BC)	00001010					Load A with (BC)
LD A,(DE)	00011010					Load A with (DE)
LD A,I	11101101 01010111	●	●	●	●	Load A with I
LD A,(nn)	00111010 n n					Load A with location nn
LD A,R	11101101 01011111	●	●	●	●	Load A with R
LD (BC),A	00000010					Store A to (BC)
LD (DE),A	00010010					Store A to (DE)
LD (HL),n	00110110 n					Store n to (HL)
LD dd,nn	00dd0001 n n					Load register pair with nn
LD dd,(nn)	11101101 01dd1011 n n					Load register pair with location nn
LD HL,(nn)	00101010 n n					Load HL with location nn
LD (HL),r	01110 r					Store r to (HL)
LD I,A	11101011 01000111					Load I with A
LD IX,(nn)	11011101 00101010 n n					Load IX with nn
LD IX,nn	11011101 00100001 n n					Load IX with location nn
LD (IX+d),n	11011101 00110110 d n					Store n to (IX+d)
LD (IX+d),r	11011101 01110 r d					Store r to (IX+d)

LD IY,nn	11111101	00100001	n	n	Load IY with nn
LD $IY,(nn)$	11111101	00101010	n	n	Load IY with location nn
LD $(IY+d),n$	11111101	00110110	d	n	Store n to $(IY+d)$
LD $(IY+d),r$	11111101	01110 r	d		Store r to $(IY+d)$
LD $(nn),A$	00110010	n	n		Store A to location nn
LD $(nn),dd$	11101101	01dd0011	n	n	Store register pair to loc'n nn
LD $(nn),HL$	00100010	n	n		Store HL to location nn
LD $(nn),IX$	11011101	00100010	n	n	Store IX to location nn
LD $(nn),IY$	11111101	00100010	n	n	Store IY to location nn
LD R,A	11101101	01001111			Load R with A
LD r,r'	01 r r'				Load r with r'
LD r,n	00 r 110	n			Load r with n
LD $r,(HL)$	01 r 110				Load r with (HL)
LD $r,(IX+d)$	11011101	01 r 110	d		Load r with $(IX+d)$
LD $r,(IY+d)$	11111101	01 r 110	d		Load r with $(IY+d)$
LD SP,HL	111111001				Load SP with HL
LD SP,IX	11011101	11111001			Load SP with IX
LD SP,IY	11111101	11111001			Load SP with IY
LDD	11101101	10101000			Block load, f'ward, no repeat
LDDR	11101101	10111000			Block load, f'ward, repeat

Mnemonic	Format	S	Z	P/V	C
LDI	11101101 1010000			●	
LDIR	11101101 10110000			0	
NEG	11101101 01000100	●	●	●	●
NOP	00000000				
OR r	10110 r	●	●	●	0
OR n	11110110 n	●	●	●	0
OR (HL)	10110110	●	●	●	0
OR (IX+d)	11011101 10110110 d	●	●	●	0
OR (IY+d)	11111101 10110110 d	●	●	●	0
OTDR	11101101 10111011		●	●	0
OTIR	11101101 10110011		●	●	
OUT (C),r	11101101 01 r 001	●	●	●	
OUT (n),A	11010011 n	●	●		
OUTD	11101101 10101011		●	●	
OUTI	11101101 10100011		●	●	
POP IX	11011101 11100001	●			
POP IY	11111101 11100001	●			
POP qq	11qq0001				
PUSH IX	11011101 11100101	●			

Description

Block load, b'ward, no repeat

Block load b'ward, repeat

Negate A (two's complement)

No operation

A OR r to A

A OR n to A

A OR (HL) to A

A OR (IX+d) to A

A OR (IY+d) to A

Block output, b'ward, repeat

Block output, f'ward, repeat

Output r to (C)

Output A to port n

Block output, b'ward, no rpt

Block output, f'ward, no rpt

Pop IX from stack

Pop IY from stack

Pop qq from stack

Push IX onto stack

PUSH IY	11111101	11100101	Push IY onto stack	
PUSH qq	11qq0101		Push qq onto stack	
RES b,r	11001011	10 b r	Reset bit b of r	
RES b,(HL)	11001011	10 b 110	Reset bit b of (HL)	
RES b,(IX+d)	11011101	11001011 d 10 b 110	Reset bit b of (IX+d)	
RES b,(IY+d)	11111101	11001011 d 10 b 110	Reset bit b of (IY+d)	
RET	11001001		Return from subroutine	
RET cc	11 c 000		Return from subroutine if cc	
RETI	11101101	01001101	Return from interrupt	
RETN	11101101	01000101	Return from non-maskable int	
RL r	11001011	00010 r	Rotate left thru carry r	
RL (HL)	11001011	00010110	Rotate left thru carry (HL)	
RL (IX+d)	11011101	11001011 d 00010110	Rotate left thru carry (IX+d)	
RL (IY+d)	11010101	11001011 d 00000110	Rotate left thru carry (IY+d)	
RLA	00010111		Rotate A left thru carry	
RLC r	11001011	00000 r	Rotate left circular r	
RLC (HL)	11001011	000000110	Rotate left circular (HL)	
RLC (IX+d)	11011101	11001011 d 000000110	Rotate left circular (IX+d)	
RLC (IY+d)	11111101	11001011 d 000000110	Rotate left circular (IY+d)	
RLCA	00000111		Rotate left circular A	

Mnemonic	Format	Description	S	Z	P/V	C
RLD	11101101 01101111	Rotate bcd digit left (HL)	●	●	●	●
RR r	11001011 00011 r	Rotate right thru carry r	●	●	●	●
RR (HL)	11001011 00011110	Rotate right thru carry (HL)	●	●	●	●
RR (IX+d)	11011101 11001011 d 00011110	Rotate right thru cy (IX+d)	●	●	●	●
RR (IY+d)	00011110 11001011 d 00011110	Rotate left thru cy (IY+d)	●	●	●	●
RAA	00011111	Rotate A right thru carry				●
RRC r	11001011 00001 r	Rotate r right circular	●	●	●	●
RRC (HL)	11001011 00001110	Rotate (HL) right circular	●	●	●	●
RRC (IX+d)	11011101 11001011 d 00001110	Rotate (IX+d) right circular	●	●	●	●
RRC (IY+d)	11111101 11001011 d 00001110	Rotate (IY+d) right circular	●	●	●	●
RRCA	00001111	Rotate A right circular				●
RRD	11101101 01100111	Rotate bcd digit right (HL)	●	●	●	●
RST p	11 + 110	Restart to location p				
SBC A,r	10011 r	A-r-CY to A	●	●	●	●
SBC A,n	11011110 n	A-n-CY to A	●	●	●	●
SBC A,(HL)	10011110	A-(HL)-CY to A	●	●	●	●
SBC A,(IX+d)	11011101 10011110 d	A-(IX+d)-CY to A	●	●	●	●
SBC A,(IY+d)	11111101 10011110 d	A-(IY+d)-CY to A	●	●	●	●
SBC HL,ss	11101101 01ss0010	HL-ss-CY to HL	●	●	●	●

[illegible]

Mnemonic

SUB (IX + d)
 SUB (IY + d)
 XOR r
 XOR n
 XOR (HL)
 XOR (IX + d)
 XOR (IY + d)

Format

11011101	10010110	d
11111101	10010110	d
10101 r		
11101110	n	
10101110		
11011101	10101110	d
11111101	10101110	d

Key:

Instruction Fields:

Condition Codes:

● = affected
 0 = reset
 1 = set
 = unaffected

Description

A-(IX + d) to A
 A-(IY + d) to A
 A EXCLUSIVE OR r to A
 A EXCLUSIVE OR n to A
 A EXCLUSIVE OR (HL) to A
 A EXCLUSIVE OR (IX + d) to A
 A EXCLUSIVE OR (IY + d) to A

C
 ●
 ●
 0
 0
 0
 0
 0

P/V
 ●
 ●
 ●
 ●
 ●
 ●
 ●

Z
 ●
 ●
 ●
 ●
 ●
 ●
 ●

S
 ●
 ●
 ●
 ●
 ●
 ●
 ●

b bit field 0-7

c condition field 0 = NZ, 1 = Z, 2 = NC, 3 = C
 4 = PO, 5 = PE, 6 = P, 7 = M

d indexing displacement +127 to -128

dd register pair: 0 = BC, 1 = DE, 2 = HL, 3 = SP

e relative jump displacement +127 to -128

n immediate or address value

pp register pair: 0 = BC, 1 = DE, 2 = IX, 3 = SP

qq register pair: 0 = BC, 1 = DE, 2 = IY, 3 = SP

r register: 0 = B, 1 = C, 2 = D, 3 = E, 4 = H, 5 = L, 7 = A

r' register: same as r

ss register pair: 0 = BC, 1 = DE, 2 = HL, 3 = SP

t RST field: Location = t*8

INDEX

- Absolute location 17, 52
- Addition, assembly-language 112-122
- Arguments, for macros 64
 - general 21, 22, 98, 111
- Array storage of machine code 110, 111
- ASCII characters, general 22, 23, 142, 185
 - working with 135-158
- ASEG pseudo-op 66
- Aspect ratio 189
- Assembling, hand 86
- Assembly, location counter 43, sequence 49
- Backspacing, on input 145
- BASIC, interfacing 94-98
 - address computation 105
- Binary/hexadecimal
 - conversion 15
- Binary search 173-176
- Buffering, characters 295, 296
- Breakpointing 83, 91
- Bubble sort 180-183
- Carries 117
- Character positions 184, 190, 191
- Characters, line printer 220
- Checkpointing 410
- Checksum 77
- CHR\$ strings, embedding
 - machine code in 106-108
- CLOSE I/O call 260
- Coding 279, 280
- Command file 87
- Comment line 13
- Comment field 13
- Conditional assembly 68
- Constants 42
- Control codes 142
- Coverision, ASCII
 - binary/hexadecimal 158
 - ASCII decimal to binary 153-155
 - binary to decimal ASCII 155-158
- Cross-reference listing 69
- CSEG pseudo-op 66
- DATA statements,
 - embedding machine code in 104-105, 108-111
- DC pseudo-op 149
- Debounce, keyboard 140, 293
- DEBUG 90-93
- Debugging 90-94, 281, 282
- Decimal/binary conversion 15
- Decimal/hexadecimal
 - conversion 15
- DEFB pseudo-op 43, 44
- DEFM pseudo-op 43-45
- DEFS pseudo-op 45, 46
- DEFW pseudo-op 43, 44
- Delimiter 69
- Desk checking 81, 82, 91, 280
- Device control blocks (DCBs) 247-250
- Displacement field 102

- Display, of characters 148
 - of input characters 150
 - of message
- Disk Assembler 53-74
- Disk Assembler, edit
 - commands 56, 57
 - files 87
 - labels 62
 - loader 61, 62, 71-74
 - macros 62
 - using 70
- Disk, characteristics 235-238
 - controller 241-245
 - device control blocks 247-250
 - drives 238-241
 - formatting 238
 - signals 240
 - TRSDOS I/O calls 251-263
 - TRSDOS organization 245-247
- Disk files, for Disk Assembler 55
- Disk input/output 235-263
- Divide operations, 127-130
- DJNZ instruction 102
- Dummy strings 108, 109
- DUMP command 93
- DSEG pseudo-op 66
- Edit buffer 55
- Editor 13, 38
- EDTASM 37-52
- Effective address 112
- ENTRY pseudo-op 62
- EQU pseudo-op 46, 47
- Expressions 48
- EXT pseudo-op 62
- Fields 12
- Flags 31, 32, 35, 41, 115, 122
- Floating-point operations 134
- Flowcharting 272, 273
- Format differences, between EDTASM and Disk Assembler 54
- Free format 39
- Graphics, animation 199
 - codes 188
 - drawing patterns and figures 197-201
 - line drawing 193-196, 203-210
 - processing 184-210
 - random points 202
- Hamming code 266
- Hand assembling 86
- Hashing technique 401
- Hexadecimal 15
- Immediate value 28, 33
- INIT I/O call 256, 257
- Input buffer 145
- Input subroutines 144-147
- Input/output programming 211-217
- Input/output, address 213
 - cassette 221-227
 - cassette music 227-233
 - I/O mapped 212-214
 - memory-mapped 215-217, 242
 - port addresses 214
 - printer 217-221
- Instruction set 9-10
- Instruction times 270
- JR instructions 102
- Keyboard, bounce 136, 137
 - scan 135-143
- KILL I/O call 260
- Listing 10, 60
- Listing file 59
- Loader, Disk Assembler 71-74
- Location 17
- Lower case 186

- Machine code, embedding in
 - BASIC 99-111
 - general 11, 14, 99
- Machine language 10, 13
- Macros, Disk Assembler
 - 62-65
- Memory allocation 78-81,
 - 87-89
- Mnemonic 10, 11, 40, 41
- Modulus operation 132
- Morse Code Generator
 - Program 285-331
- Multiple-precision operations
 - 121, 134
- Multiply operations 123-126
- Object code 50
- Object file 77
- Op code 12, 38
- Operand 12
- Operation code 12
- Operators 48
- OPEN I/O call 251, 253, 254
- ORG pseudo-op 42
- Overflow 129, 130
- Pages, Disk Assembler 58
- Parameter list 98
- Patching 83-87, 93
- Periodicals, programming
 - 268
- Permutations 340-342
- Pixels 189-192
- Pointer 23
- POSN I/O call 261-263
- Preliminary specification
 - 268-271
- Printer input/output 217-221,
 - 326
- Program counter 18, 102
- Program design 271-278
- Program sections 54, 66
- Pseudo-ops 38, 42
- Pseudo-random number
 - generation 131
- P/V flag 114, 115
- Random number generation
 - 130, 132, 321
- Ranges, of lines 55
- READ I/O call, 254-256
- Registers 23, 25
- Relative location 17
- Relocatability 50-52, 54, 61,
 - 93, 99-103
- Relocatable object file 59
- Research 266-268
- Restoring divide 127, 128
- Scaling 205, 206
- Scrolling, 150-152
- Searching 173-176
- SET/RESET 192
- Shift and add multiply, 124,
 - 125
- Signed multiplies and divides
 - 129
- Simulating text 287
- Single stepping 91
- Sorting 177-183
- Source code 11, 39
- Source file, creating on Disk
 - Assembler 58
 - general 14, 55
- Source line 59
- Stack operations 18-20, 28,
 - 78, 79
- Subcommands, Disk
 - Assembler editor 57
- Subroutine 11, 17, 111
- Subtract operations 116-122
- Successive addition multiply
 - 123
- Successive subtract divide
 - 127
- Symbol table 28, 46
- Symbolic code 10
- Syntax, assembler 39-41
 - editor 38
 - input subroutine 144

SYSTEM tapes 75-78	Testing, 282
T-BUG 78, 81-86	Tic-Tac-Toe Program 333-410
Table size, automatic 168	Tone generation, through
Tables, fixed-length entry,	cassette port 227-233, 291
fixed length 159-161	Trace 34
fixed-length entry,	TRSDOS I/O calls 251-263
variable length 162, 163	Two-buffer sort 177-180
jump 165-169	Unsigned multiplies and
ordered 172	divides 129
scanning 169-171	USR calls 94-97
searching 173-176	Word format 44, 45
sorting 177-183	WRITE I/O call 257, 258
variable-length entry,	Z flag 21
variable length 164, 165	Z-80 9
working with 159-183	Z-80 registers 23, 25

Radio Shack

A Division of Tandy Corporation
Fort Worth, Texas 76102